Nantucket®

Clipper™

WELCOME ABOARD

Welcome to Clipper™

We are pleased that you have chosen to work with Clipper, the premier dBASE™ compiler. Clipper continues to represent Nantucket's ® philosophy that software products should give you the independence to develop powerful programs without imposing restrictions. This is one reason why we have designed Clipper with an open architecture that gives you room to add program enhancements that solve your programming needs.

Combine Nantucket's philosophy with new features, commands and functions, and you will see that Clipper is an indispensible tool for developers and professionals. Nantucket feels this level of sophistication is a requirement for any serious developer in today's competitive database marketplace.

Rest assured that our involvement with you does not stop once you purchase Clipper. Nantucket offers several support services and publications to help you utilize Clipper to its fullest potential.

We consider your ideas and suggestions on our product and services crucial to Clipper's development, one reason why Clipper continues to be the industry's number one compiler. The enclosed reader comment cards allow you to comment on our software and manual. Please take the time to complete and return them to Nantucket.

Nantucket Corporation is pleased to have this opportunity to provide you with an exceptional product. Please contact us if we can be of further assistance.

We look forward to working with you in the future.

Sincerely,

George A. Juarez
Chairman and
Chief Executive Officer

REGISTRATION CARD

At the back of this manual you will find the following:

- The Clipper Registration Card
- A Reader Comment Card

Before you do anything else, fill out and mail the Clipper Registration Card. This allows you to receive valuable product and update information, as well as a complimentary copy of Nantucket News. Below it is a card for your comments and suggestions for improving the contents of this manual or suggested enhancements to Clipper. This card may be mailed at any time.

We have included a file called READ_ME.1ST on your Clipper diskette. This file is the first thing you should read, since it contains the most current information regarding the Clipper Compiler. Any last minute changes or enhancements to the manual are documented in this file.

COPYRIGHT NOTICE

Copyright © 1985, 1986, 1987 Nantucket Corp. All rights are reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any other language or computer language in whole or in part, in any form or by any means, whether it be electronic, mechanical, magnetic, optical, manual or otherwise, without the prior written consent of Nantucket Corporation, 12555 W. Jefferson Blvd., Suite 300, Los Angeles, California 90066.

DISCLAIMER

The Clipper User Manual is printed in the U.S.A. Nantucket believes that the information contained in the manual is correct. However, Nantucket reserves the right to revise the manual and make periodic changes to the content without a direct or inferred obligation to Nantucket to notify any person of such revision or changes. Nantucket does not assume responsibility for the use of the manual, nor for any patent infringements or other rights of third parties who may use the manual or the software contained herein.

The software contains valuable trade secrets and proprietary information. Unauthorized use of the manual or the software can result in civil damages and criminal prosecution. Please see Appendix B for additional information.

COPYRIGHT, TRADEMARK AND SERVICE MARK NOTICES

CLIPPER is a trademark of Nantucket Corporation

NANTUCKET is a registered trademark of Nantucket Corporation

NANTUCKET SUPPORT is a service mark of Nantucket Corporation

dBASE, dBASE III PLUS and ASHTON-TATE are trademarks of Ashton-Tate, Inc.

dBASE II is a registered trademark of Ashton-Tate, Inc.

IBM, PC-DOS and AT are registered trademarks of Internation Business Machines Corporation

Microsoft is a registered trademark of Microsoft Corp.

Microsoft C Runtime Library Routines, Copyright Microsoft Corporation, 1984, 1985, 1986, 1987. All rights reserved.

MS-DOS and MASM are trademarks of Microsoft Corp.

PC/XT is a trademark of International Business Machines Corporation

PLINK86-Plus is a trademark of Phoenix Technologies, Ltd.

The Source is a registered service mark of Source Telecomputing Corporation

Some of the trademarks of other companies mentioned in this manual appear for identification purposes only.

CREDITS

Product Development

Brian Russell

Richard McConnell

Rick Spence

David Dodson

Kevin Shepherd

Jon Rognerud

Product Documentation

Vickie Deinhard

Ira Emus

Renee Gentry

Steve Hillbourne

Fred Ho

Phil Kimble

Ray Love

David Morgan

Don Powells

Vukan Ruzic

Christopher White

CONTENTS

NANTUCKET CLIPPER

CHAPTER 1

Introduction To Clipper **1-1**

Description of the Contents of the Manual 1-2
The Clipper Compiler 1-4
Differences Between an Interpreter and a Compiler . . . 1-4
Executing the Clipper Compiled Program on any PC/MS DOS Computer 1-7

CHAPTER 2

Installing Clipper On Your Computer **2-1**

Computer System Requirements 2-2
The Most Recent Clipper Documentation 2-3
Installing Clipper 2-3
To Quickly Compile and Link 2-4

CHAPTER 3

The Distinguishing Features of Clipper **3-1**

Clipper Advantages 3-2
Clipper Features 3-4
Clipper Enhancements - Commands 3-10
Clipper Enhancements - Functions 3-12
Clipper Utilities 3-15
Modifications to dBASE III PLUS Applications 3-16

CHAPTER 4

The Clipper Language **4-1**

Technical Specifications 4-2
Files 4-3
File Aliases 4-5
Fields 4-6
Constants 4-7
Memory Variables 4-8
Operators 4-10
Expressions 4-14
Syntax Rules 4-15
Macro Substitution 4-17
User-Defined Functions 4-18
Establishing Multiple Relations to a File 4-19
Designing Customized Help 4-20
Dividing By Zero 4-24
dBASE III PLUS Compatible Indexes 4-25
Key Tables for Full Screen Operations 4-26

CHAPTER 5

Commands **5-1**

Conventions Used in Commands and Functions 5-2
Summary of Clipper Commands 5-6
Command Descriptions and Examples 5-21

CHAPTER 6

Functions **6-1**

Summary of Clipper Functions 6-2
Function Descriptions and Examples 6-14

CHAPTER 7

Compiling and Linking Your Programs 7-1

What Does the Clipper Compiler Do? 7-2
Compiling Your Programs 7-2
What Does the Linker Do? 7-7
Linking Your Programs with the DOS Linker 7-7
Linking Your Programs with PLINK86-Plus 7-8
Using Batch Files 7-14
Linking Programs with the Clipper Debugger 7-15
Linking Programs with Optional Screen Drivers 7-16
Linking Programs with Other Function Libraries 7-17
Overlays 7-18

CHAPTER 8

The Clipper Debugger 8-1

What is the Clipper Debugger? 8-2
How to Include the Debugger In Your Programs 8-2
Using the Clipper Debugger 8-3
Navigation Within the Debugger 8-4
Menu Bar Selections 8-6
Invoking the Debugger 8-13

CHAPTER 9

The Runtime Environment 9-1

The DOS Command Processor - COMMAND.COM . . . 9-2
DOS Files and Buffers 9-2
ANSI Terminal Support 9-3
Computer Memory Usage 9-4
Modifying the Runtime Environment 9-6

CHAPTER 10

Using Clipper With a Local Area Network 10-1

What Is a Local Area Network 10-2
Clipper Features 10-2
Network Commands 10-3
Network Functions 10-4
Compatibility fo Local Area Networks With Clipper 10-4
Programming For a Local Area Network Environment . . . 10-5
Effects of the Network Environment on Files 10-11
Effects of the Network Environment on Commands 10-13
Locks.prg Source Code 10-13

CHAPTER 11

The Extend System 11-1

CHAPTER 12

Clipper Utility Programs 12-1

The DBU.EXE Program 12-2
The RL.EXE Program 12-6
The Index Program 12-10
The Line Program 12-11
The Make Program 12-12

APPENDIX A

Nantucket Support A-1

APPENDIX B

Nantucket Software License Agreement B-1

APPENDIX C

dBASE III PLUS Commands and Functions Not Supported By Clipper C-1

APPENDIX D

Clipper Compiler Error Messages D-1

APPENDIX E

PLINK86-Plus Error and Warning Messages E-1

APPENDIX F

Runtime Error Messages F-1

APPENDIX G

An ASCII Chart and the INKEY() Return Values G-1

APPENDIX H

Reserved Words H-1

APPENDIX I

PLINK86-Plus Commands I-1

APPENDIX J

Sample Programs J-1

GLOSSARY

Glossary I

INDEX

Index Index-1

1 Introduction To Clipper

Chapter 1 contains an introduction to the Clipper compiler. Topics include:

- A description of the contents of each chapter and appendix of the manual.
- A discussion defining the differences between an interpreter and a compiler.

Chapters

This manual is organized into 12 Chapters and 10 Appendices, which are briefly described below.

- Chapter 1 contains an introduction to Clipper, including an overview of the manual and an explanation of how an interpreter and a compiler work.
- Chapter 2 contains the instructions for installing Clipper and the computer system requirements for installing, compiling and executing Clipper programs.
- Chapter 3 describes the advantages of Clipper, including additional enhancements that are available only with Clipper.
- Chapter 4 contains information about the Clipper implementation of the dBASE language.
- Chapter 5 contains a detailed reference of all Clipper commands.
- Chapter 6 contains a detailed reference of all Clipper functions.
- Chapter 7 contains instructions for compiling, linking and running your programs, including path considerations, running the Clipper compiler, linking programs using either the PLINK86-Plus or the DOS linker and the use of program overlays.
- Chapter 8 contains instructions for using the Clipper Debugger.
- Chapter 9 contains information and instructions regarding Clipper and the DOS environment. Topics cover the DOS command processor, DOS files and buffers, ANSI terminal support and computer memory usage.
- Chapter 10 contains information and instructions for writing Clipper applications for use in a Local Area Network environment.

- Chapter 11 contains information detailing the power and use of the Clipper Extend System.
- Chapter 12 contains instructions for using the Clipper utility programs.

Appendices

- Appendix A describes how to obtain technical support and assistance in using Clipper.
- Appendix B contains the Nantucket Software License Agreement.
- Appendix C contains a list of dBASE III PLUS commands not supported by Clipper.
- Appendix D contains the compilation error messages displayed by Clipper.
- Appendix E contains the PLINK86-Plus error messages.
- Appendix F contains error messages that may be displayed during the execution of the application program.
- Appendix G contains a list of INKEY() values and an ASCII chart.
- Appendix H contains a reserved word list.
- Appendix I contains PLINK86-Plus commands.
- Appendix J contains programming examples.

Clipper is a stand-alone system containing all facilities required for application development. With Clipper you can create and compile programs without the assistance of additional software, other than a text editor. After error-free Clipper programs have been compiled and linked, the executable (.EXE) file can be run directly on any computer that supports PC/MS-DOS version 2.0 or greater (PC/MS-DOS version 3.1 or greater for network applications).

Overview Of An Interpreter

To create a program using an interpretive language, you create a file containing the various commands and functions comprising your program. Each line of code is executed on an individual basis every time that you run the program. The interpreter "steps" through each line of code and, if no errors are detected, converts the line into code that can be understood by the computer (machine-language code).

If an error is found prior to the conversion, execution ceases and an error message is displayed on the screen. You must then correct the error and rerun the program. This process is repeated until the program is error-free. Unfortunately, once the program is perfected, the program is still subject to the same time-consuming process of checking each individual line each time it is run.

Finding and Correcting Errors

If a syntax error is found on any line, the execution ceases and the interpreter displays an appropriate error message on the screen. You must then return to the text editor, correct the problem and again execute the program under the interpreter's control. This immediate feedback can be a helpful debugging aid during initial program development.

However, after you correct the first problem, the interpreter again analyzes, converts and executes each line starting from the first line. Each time an error is encountered, execution ceases and an error message is displayed. This process is repeated until the program has been perfected.

EXECUTING THE CLIPPER COMPILED-PROGRAM ON ANY PC/MS-DOS COMPUTER

With Clipper-compiled programs, you are no longer constrained to running the program on the computer where it was originally compiled and linked. You do not need an interpreter, the original program (source code), the object code (.OBJ) file or even a copy of Clipper. You do need a computer that includes:

- PC/MS-DOS Version 2.0 or greater for single-user application
- PC/MS-DOS Version 3.1 or greater for network applications
- IBM PC or ANSI terminal I/O support
- The executable (.EXE) file
- Any database, index, report form, label or memory files required by the executable file

2 Installing Clipper On Your Computer

Chapter 2 contains instructions for installing Clipper on your computer. Topics covered include:

- Computer system requirements for compiling and executing programs with Clipper.
- Reference to the most recent Clipper documentation.
- Instructions for installing Clipper on systems with a hard disk drive.
- Quick compiling and linking using a batch file.

Compiling Programs With Clipper

Clipper requires that you have an IBM PC, XT or AT, or a 100% IBM compatible computer. The computer must include the following features:

- PC-DOS or MS-DOS Version 2.0 or greater for single-user applications
- PC-DOS or MS-DOS Version 3.1 or greater for network applications
- 256K, or more, of Random Access Memory (RAM)
- 1 diskette and 1 hard disk drive
- Useful options include a printer, an 8087 (or 80287) math co-processor chip, and communications capability that will permit access to The Source℠ and/or BIX™.

Executing Clipper-Compiled Programs

Clipper-compiled programs can be run on any computer that has the following attributes:

- PC-DOS or MS-DOS Version 2.0 or greater for single-user applications
- PC-DOS or MS-DOS Version 3.1 or greater for network applications
- 256K, or more, of Random Access Memory (RAM)
- An IBM PC, XT or AT (or a computer that is 100% compatible), or
- A computer that has ANSI terminal support

THE MOST RECENT CLIPPER DOCUMENTATION

A file called READ_ME.1ST is included on your Clipper System Disk. This file contains the most current information regarding the Clipper compiler, including last minute changes, corrections to the manual, usage information, and a list of files contained on your Clipper diskettes. It is recommended that you print a copy of the file for future reference.

INSTALLING CLIPPER

The diskettes supplied with Clipper should not be used to compile and link programs. You should first copy the Clipper program onto your hard disk, then copy Clipper onto a floppy diskette as a back-up.

Installation Procedure

From the root directory create a sub-directory that will contain the Clipper files. The following DOS command will create a sub-directory called "CLIPPER".

C>MD CLIPPER

Once you have created the sub-directory, select the new sub-directory by entering:

C>CD CLIPPER

Insert the Clipper System Disk in drive A and select the A drive by entering "A:" Enter the following command:

A>CLIPCOPY <drive>

On the above command line, "<drive>" is the disk drive identification letter. The CLIPCOPY.BAT file is a DOS batch program which will copy the required files onto your hard disk.

Chapter 9 contains helpful information on setting up AUTOEXEC.BAT and CONFIG.SYS files. Use this as a guide when creating your DOS environment.

Clipper contains a useful batch file to assist in compiling and linking your programs. Use CL.BAT to compile and link your application on a hard disk.

It is recommended that you read Chapter 7, "*Compiling and Linking Your Programs*". However, for a quick example of the compiling and linking process, enter the following command:

C>CL INDEX

"INDEX" is a program which allows you to create Clipper compatible indexes for your applications. Prior to using INDEX.PRG you must compile and link it. INDEX.PRG is included on your Clipper disk.

3 The Distinguishing Features of Clipper

Chapter 3 discusses the enhancements of Clipper and the differences between Clipper and dBASE III PLUS. Topics covered include:

- Clipper advantages.
- Clipper features.
- Clipper enhancements - commands.
- Clipper enhancements - functions.
- Clipper Utilities.
- Modifications to dBASE III PLUS applications.

Clipper offers many unique and powerful tools to database programmers. This chapter is an overview of those features. Each feature is covered in detail in subsequent chapters of this manual. Clipper advantages include:

- Dramatically faster execution.
- Absolute security of the source code.
- Ability to execute Clipper-compiled and linked programs on any computer that supports PC/MS-DOS version 2.0 or greater for single-user applications or DOS 3.1 or greater for network applications.
- Ability to distribute Clipper-compiled programs without royalties.
- Enhanced networking capability without royalty or licensing fees.
- Ability to place procedures and functions in the same file as the calling program.
- A group of functions and operators which can handle strings up to 64K in length.
- A choice of faster Clipper compatible index files or dBASE III PLUS compatible index files.
- Access to low-level DOS files and devices.
- Use of up to 2048 active memory variables.
- Use of up to 1024 fields per database file.
- Use of 255 open files (with DOS 3.3).
- Ability to create user-defined functions.
- Verification of source code during each compile.
- Identification of all program errors at one time.
- Use of up to eight "child" relations to a "parent" file.

- Ability to call an unlimited number of outside programs written in C or Assembler.
- Ability to treat memo fields as string values.
- Use of recursive or nested macros.
- Use of nine characters in the @...BOX command.
- Ability to create an empty structure extended file without an existing database file.
- Use of the PUBLIC variable CLIPPER to allow Clipper-enhanced programs to run interpreted.
- Ability to execute loops using the BASIC-like FOR...NEXT command.
- Ability to return a logical true if a USE, USE...EXCLUSIVE, or APPEND BLANK fails in a network environment.
- Ability to DECLARE and manipulate arrays.
- Use of enhanced memo field functions.
- Ability to APPEND BLANKs to a shared file.
- Ability to position the record pointer at, and display, a record locked by another user.
- Allowance for an unlimited number of users to access the same database (see Chapter 10 for network information).
- Ability to return the text of the computer name in a network.
- Ability to provide context-sensitive help facilities, such as data entry instructions, when the user presses F1.
- Use of utilities that are accessed directly from the operating system to create database and index files, labels and reports.
- Use of complete debugging facilities.

General Enhancements

Clipper supports a number of enhancements that affect the behavior and syntax of most commands and functions. These are intended to make programming easier and extend the power of your applications.

User-defined Functions

Clipper supports the dBASE III PLUS programming language predefined functions, such as EOF() and CHR(). In addition, you may create your own functions, which, once defined, may be used anywhere in the program. You can build function libraries which are incorporated into the program during the linking process to make these functions available. See Chapters 4 and 6 for further examples of user-defined functions.

Begin a Command Line with a Function

Clipper permits you to begin a command line with a function. This can be very useful with functions that cause an action and either do not return a value or return a nonessential value. For example:

```
? "Copyright 1986, Emulsified Software"
INKEY(1)          && Wait 1 second.
```

Enhanced Use of Macros

As with the dBASE III PLUS programming language, you can use macros in place of constants, variable names, full expressions (including functions and operators) and literals. In addition, Clipper allows the use of macros for conditions in DO WHILE statements, as well as the use of recursive macros.

Extended Use of Expressions

In Clipper, you can now specify an expression wherever you could, formerly, only specify a literal identifier as an argument. This applies generally to SETs and commands that open files. To do this, you bound the expression with parentheses. For example, you can now USE a database file with the following syntax:

```
USE (<expC>)
```

instead of

```
USE <filename>
```

This is a global capability, so refer to the specific command entry in Chapter 5 for more information.

DOS File Functions

Clipper supports low-level access to DOS files and devices with a number of new functions. These functions allow you to create, open, close, read, write, seek, and trap errors in DOS files. See Chapter 6 for more information.

Enhanced Screen Interface

Clipper provides a number of features that enhance your ability to create sophisticated interfaces for users.

Light-bar Menus

In Clipper, there are two facilities for light-bar menus. You have first, the combination of @...PROMPT, SET MESSAGE, and MENU to create light-bar menus with any orientation of menu pads you wish. @...PROMPT paints each menu pad and defines an associated message. SET MESSAGE TO defines the screen row for the message display. MENU TO invokes the light-bar selection mechanism. Pressing Return or the first character of the menu pad selects the choice, returning its position in the list of choices as a numeric value.

A second light-bar facility is available using the ACHOICE() function to create a vertical list light-bar menu. To execute an ACHOICE() menu, you define the menu window coordinates and an array of choices. The choice is selected either by positioning the light-bar or by pressing the first key of the choice. Pressing Return confirms the choice, returning its position in the list of choices also as a numeric value.

If the number of choices is greater than the number of rows in the menu window, ACHOICE() will scroll the choices.

SAVE and RESTORE Screens

Clipper allows you to SAVE and RESTORE SCREENs TO memory variables including array elements.

Keyboard Control

There are a number of enhanced features that allow you to control the keyboard more effectively. LASTKEY() returns the last key pressed in a "wait state" including READ, MENU TO, ACCEPT, INPUT, and WAIT. The KEYBOARD command can stuff the keyboard with one or more characters from a "wait state." You can reassign keys using SET KEY TO so that when you press a key within a "wait state" control passes to a procedure. When the procedure terminates, control then returns to the "wait state." INKEY() is enhanced with an optional time delay allowing you to pause execution for a specified time or wait indefinitely for a key press.

Turn the Cursor On and Off

With SET CURSOR, Clipper allows you to turn the display of the cursor on or off.

Custom Help

Clipper enables you to include custom-designed help by compiling any procedure or program named Help into your application. Help is then invoked with F1 and available from any "wait state" including READ, MENU TO, ACCEPT, INPUT, and WAIT. See Chapter 4 for more details.

Validation of GETs

With the VALID clause of @...GET, Clipper allows you to validate individual GETs without terminating the READ. VALID works by executing a logical expression when you attempt to leave the GET. If the expression returns true (.T.), the current GET terminates; otherwise the cursor remains in the GET until the VALID expression returns true (.T.) or you press Esc.

VALID also supports user-defined functions. With this facility, you can perform various levels of validation. This includes look-ups in another work area or pop-up menus using MENU TO or ACHOICE(). Additionally, you can assign a new value to a GET variable or field and upon RETURNing from the user-defined function, the GET display and contents are updated with the new value.

Undo in GETs

When you are editing a GET, you can undo any recent changes you have made by pressing Ctrl-U. For example, if the initial GET value is "TED" and you change it to "FRED," pressing Ctrl-U returns the GET to the original value of "TED." Once you have left the current GET, however, you cannot undo your previous changes.

User-defined Box Character

In addition to @...TO which draws boxes with single or double lines, Clipper also supports @...BOX which allows you to define the border characters of the box along with a fill character.

Enhanced Memory Variables

Arrays

Clipper allows you to define one-dimensional arrays. Each array is counted as one memory variable and may be passed as a parameter to procedures or user-defined functions. There are, additionally, a number of array functions provided that allow you to insert and delete elements, fill an array, scan an array for a specified value, load an array with filenames from a disk directory

using wildcards, and several other capabilities. The array functions can be found in Chapter 6.

More Memory Variables

Clipper allows you to use up to 2,048 public and private memory variables (memory permitting) with each array counting as one memory variable. Arrays in turn can contain 2,048 elements.

Longer Character Strings

In previous versions, Clipper operators and functions supported character strings with a maximum length of 32K. The following have been altered to support the longer strings of 64K:

+	LOWER()
==	REPLICATE()
$	SPACE()
AT()	SUBSTR()
EMPTY()	TRIM()/LTRIM()
LEN()	UPPER()

Enhanced Network Processing

Clipper handles network files differently than other database products. In Clipper, users are barred from accessing locked files or records when writing only. An unlimited number of users can access a shared database file for reading. This allows a more efficient use of computer time. NETERR() is used to indicate that a USE, USE EXCLUSIVE or APPEND BLANK has failed in the network environment. See Chapter 10 for further information on the network environment.

Enhanced Database File Processing

Relative Seeking

Using the SET SOFTSEEK command, you can perform "relative seeking." This means that when you SEEK and a match is not found, the record pointer is positioned at the record with the next higher value in the index.

Multiple Parent-Child Relations

Clipper supports multiple parent-child relations between work areas using SET RELATION. With this capability, you can relate up to eight child work areas per parent. See Chapter 4 for further information.

TOTAL and UPDATE ON an Expression

In Clipper, you can both TOTAL and UPDATE ON key expressions. This allows you to summarize and update database files based on existing index keys.

Enhanced Memo Handling

Memo fields can be assigned to character memory variables. This allows you to perform searches, concatenations, REPLACEs of memo fields and any other operations permitted on character strings.

Formatting Functions

In order to report memo fields more effectively, Clipper provides a number of formatting functions. Using MLCOUNT() and MEMOLINE(), for example, you can extract word-wrapped strings from long strings or memo fields and then print them. This allows you to print memo fields, precisely controlling the page-layout.

Memo Editor

To allow the entry and editing of memo fields or long strings, Clipper provides MEMOEDIT(), a text editing function that can operate in a window. MEMOEDIT() has a standard complement of editing features including insertion, deletion, and word-wrapping.

See Chapter 6 for more information about the memo functions.

Enhanced Parameter Passing

Variable Number of Parameters

In Clipper, the number of parameters you pass to a program, procedure, or user-defined function is not required to match the number of variables specified as arguments of the PARAMETERS command. In order to make this manageable, Clipper provides PCOUNT() which returns the number of actual parameters passed.

DOS Command Line

In addition to passing parameters to a calling program, you can also pass parameters from the DOS command line. The difference is that all parameters passed are treated as character strings. See the PARAMETERS command in Chapter 5 for more information.

Enhanced Operators

In addition to the standard operators, Clipper supports a number of enhanced operators which are shown in the following table.

Table 3-1 Enhanced Operators

Relational		Mathematical	
==	Exactly equal	%	Modulus
!=	Not equal		
!	Not		

Miscellaneous Enhancements

PUBLIC Clipper

You can specify a PUBLIC variable named Clipper in your programs. This allows you to include Clipper enhancements, such as the additional commands, in your dBASE III PLUS programs. If the expression is properly constructed, you will still be able to run the program under interpretive dBASE III PLUS. The PUBLIC command is discussed more thoroughly in Chapter 5.

Clipper Debugger

The Clipper Debugger is a powerful tool designed to assist in locating program errors. Its window orientation enables you to see the entire program environment while checking the program status, setting break points, viewing the program structure, or changing the value of a memory variable or expression. The highly interactive Debugger is menu-driven which allows you to easily move around within your program while using the Debugger options. See Chapter 8 for more information.

A complete syntax description and example of each command can be found in Chapter 5.

@...BOX	With a single statement, you can draw a box on the screen and specify a different character for each side, each corner, and the interior of the box.
@...PROMPT	Used to place menu selections on the screen.
@...SAY...GET ...VALID <expL>	Allows the program to check the validity of user input before proceeding to the next GET.
DECLARE	Creates an array of <expN> elements. These elements can be of mixed types in their usage.
EXTERNAL	Used to declare a symbol to the linker. Procedures that have been declared as EXTERNAL can then be placed into overlays and still be called with a macro.
FOR...NEXT	The FOR...NEXT command permits you to accomplish a looping operation for a range of expressions, where you may optionally increment or decrement the range expression. The STEP option allows you to specify the amount of increment or decrement.
KEYBOARD	Stuffs the keyboard input buffer with the specified string. Each time the KEYBOARD command is executed, the type-ahead buffer is cleared.
MENU TO	Invokes the light-bar menu allowing you to navigate the highlight bar between PROMPTs and returning the position of the selected PROMPT into the specified memory variable.

RESTORE SCREEN	Used in conjunction with the SAVE SCREEN command to avoid repainting the original screen.
SAVE SCREEN	Writes the current screen to a buffer or a variable. This command is used in conjunction with the RESTORE SCREEN command to eliminate the need to repaint the original screen.
SET CURSOR	Allows user to turn the flashing cursor on and off.
SET KEY	Allows a procedure to be executed from any "wait state" when a designated key is pressed, where a "wait state" is any command that pauses program execution. (See Chapter 5 for "wait state" examples.)
SET SOFTSEEK	Allows for "relative" seeking. If a specified record is not found, the record pointer is positioned at the logical record immediately following where the specified record should have been.
SET WRAP	Allows circular wrapping within MENU TO.

A complete syntax description and example of each function can be found in Chapter 6.

ALIAS()	Returns the alias of a SELECT area. If a parameter is not specified, the current alias is returned.
DTOS()	Returns a string in yyyymmdd format. This function can be used to index the concatenation of a date and a character expression.
EMPTY()	Returns a logical true if a character expression is a null string or all spaces, a numeric expression = 0, a date expression = empty date, or a logical expression returns false (.F.).
FCLOSE()	Closes a DOS file.
FCOUNT()	Returns the number of fields in the current database.
FCREATE()	Creates a DOS file.
FERROR()	Returns DOS error codes for file operations.
FOPEN()	Opens a DOS file.
FREAD()	Reads characters from a DOS file and returns a count of the characters read.
FREADSTR()	Reads characters from a DOS file and returns a string.
FSEEK()	Sets the file pointer of a DOS file to a new position.
FWRITE()	Writes to a DOS file.
HARDCR()	Replaces all soft carriage returns (ASCII character 141) with hard carriage returns

	(ASCII character 13) within a given character expression.
INDEXEXT()	Returns the extension of the type of index currently in use (.ntx or .ndx).
INDEXKEY()	Returns the key expression of an index.
INDEXORD()	Returns the current SET ORDER TO number.
LASTKEY()	Returns the numeric value of the ASCII code for the last key pressed, including control keys.
LOCK()	Attempts to lock a database record. Returns true (.T.) if the lock is successful.
MEMOEDIT()	Allows editing of memos, fields and character strings.
MEMOLINE()	Returns a formatted line from a character expression or memo field.
MEMOREAD()	Returns the specified disk file as a character string.
MEMORY()	Returns the available free pool memory in K bytes.
MEMOTRAN()	Returns a character string with the carriage return and line feed characters replaced.
MEMOWRIT()	Writes a character string to a specified disk file, and returns true (.T.) if successful.
MLCOUNT()	Returns the number of lines in a character expression or memo field.
NETERR()	Returns true (.T.) if a USE, USE...EXCLUSIVE, or APPEND BLANK fails in a network environment.

NETNAME()	Returns the text of the computer name. If the computer name was not set, it will return a null string.
PCOUNT()	Returns the number of parameters passed from the command line or from another procedure.
PROCLINE()	Returns the source code line number of the current program or procedure being executed.
PROCNAME()	Returns the name of the current program or procedure that is being executed.
READVAR()	Returns the name of the current GET/MENU variable, or a null string if none is pending.
SCROLL()	Allows a designated section of the screen to be scrolled up and down or cleared.
SECONDS()	Returns the system time as <seconds>.<hundredths>. The value returned is the number of seconds elapsed since midnight and is based on a twenty-four hour clock, in a range from 0 to 86399.
SELECT()	Returns the numeric value of the currently selected area.
SETPRC()	SETS the internal PROW() and PCOL() to the specified values.
UPDATED()	Returns a logical true if the last read changed any data in the associated GETs.

CLIPPER UTILITIES

The distribution disk contains special utilities for creating and editing files outside of Clipper. These utilities include:

DBU The DBU (database utility) program permits you to create and edit database files, index files, and data. You can also model data structures using relations and filters without writing a separate program or using an interpreter.

RL The RL utility program allows you to create and edit REPORT and LABEL FORMs. RL is written in Clipper and included with source code.

LINE The LINE program lists your programs with line numbers to either the screen or printer.

MAKE The MAKE program is used to describe source and object file dependencies to the Clipper compiler and to streamline the compile and link cycle.

For further information, see Chapter 12.

Note: The differences between Clipper and dBASE III PLUS that are described in this section refer to dBASE III PLUS version 1.1.

Clipper's syntax and logic emulate the interpreted dBASE III PLUS programming language wherever possible. There are differences, however, and where they exist it may be necessary to modify your programs before they can be compiled successfully by Clipper.

If you wish to use an existing application program that has been written in dBASE III PLUS, you will need to modify the program if one of the following three statements is true:

- The program uses one of the interpretive dBASE III PLUS commands that are not supported, or are supported in a different way, by Clipper. Appendix C contains a list of these commands.

- You wish to incorporate one of Clipper's enhancements before compiling. This chapter contains a list and explanation of each of these enhancements.

- The program uses a macro that contains a command key word or command comma, which are not allowed in Clipper. (Valid expressions, including functions, are allowed in macros.)

Please note that dBASE III PLUS programs modified to include any Clipper enhancements or commands and functions which are handled differently will NOT run on interpretive dBASE III PLUS, unless those commands and functions have been specifically incorporated in a conditional IF statement as an option associated with the PUBLIC variable CLIPPER.

4 The Clipper Language

Chapter 4 contains general information about programming with Clipper. Topics covered include:

- Clipper technical specifications
- Files used by Clipper
- File aliases
- Fields
- Constants
- Memory variables
- Operators
- Expressions
- Syntax rules
- The Macro (&)
- User-defined functions
- Establishing multiple relations to a file
- Designing customized help
- Dividing by zero
- Math processor chip
- Use of scope in commands
- dBASE III PLUS compatible index files
- Key tables for full screen operations

Database Files	Number of Records	1 billion (if diskspace allows)
Records	Structure	Sequential, Fixed Record Length
	Size	Available RAM
	Number of Fields	Available RAM
Fields	Character Type	32K Characters (bytes) Maximum
	Numeric	19 Digits (bytes) Maximum
	Date	8 Numbers (bytes) Maximum
	Logical	1 Character (byte)
	Memo	64K Characters (bytes) Maximum
Index Type		Uses B-tree branching
Memory Variables	Number	2048 Maximum
	Size	Character type - 64K (bytes) Numeric type - 19 Digits
Arrays	Number	2048 Maximum (Each counts as one memory variable)
	Elements	2048 Maximum (if memory allows)
	Element Size	Same as memory variables
Files	Open Files	255 Maximum (With DOS 3.3)
	Active Index Files	15 per Work Area Maximum
Index Keys		250 Characters per Index File Maximum
Procedures per File Maximum	Unlimited	
Other	Numeric Accuracy	18 Decimal Places (not including decimal point)

FILES

Filenames

A Clipper file referenced with an identifier consisting of two parts: the filename and the filename extension, separated by a period. The filename extensions determine the file type and are defined below.

The filename can consist of any of the letters, numbers, and symbols on the keyboard except a hyphen. Filenames must begin with a letter and cannot contain embedded blank spaces. The maximum filename length permitted is 8 characters.

Clipper requires that you give each procedure and function, including program and format files, a unique name. The Clipper compiler evaluates each filename in the same manner as other identifiers (memory variables, aliases, etc.).

Specifying Filenames in Commands

With most commands, you do not have to specify the file extension unless you have given the file an extension other than the default. If a command creates a file, the appropriate extension is automatically assigned. However, the commands COPY FILE, DELETE FILE, ERASE, RENAME, and TYPE require that you specify the filename extension.

FILES USED BY CLIPPER

Database Files

Database files contain a record and field structure definition and the actual data for each record. The default extension for data files is ".dbf"

Memo Field Files

The memo field file contains all of the memo fields for a particular database file. All memo field files have the name of the associated database file and the extension ".dbt".

Index Files

Index files are created to provide an order for records within a database file. Several index files can be associated with a particular database file. Clipper index files have one of two extensions: ".ntx" (a Clipper format index) or ".ndx" (a dBASE III PLUS compatible index).

Memory Files

A memory file contains the identification and contents of memory variables saved on disk during the execution of a Clipper program. The default extension for memory files is ".mem".

Label Form Files

The label form file contains the instructions and definitions required to print labels with the LABEL FORM command. The default extension for label files is ".lbl".

Report Form Files

The report form file contains the instructions and definitions required to print reports with the REPORT FORM command. The default extension for report form files is ".frm".

Format Files

The format file contains screen display commands (@...SAY...GET) invoked by the SET FORMAT TO command. A format file is an ASCII text file and can be created or modified with a text editor. The default extension for format files is ".fmt".

Alternate Files

An alternate file is used to capture screen output and is created with SET ALTERNATE TO. SET ALTERNATE ON causes the screen output of many commands to be sent to the specified file. The default extension for alternate files is ".txt".

Program Files

A program file contains Clipper commands for file access, data manipulation, screen handling, and process control. A program file is an ASCII text file and can be created or modified with a text editor. The default extension for program files is ".prg".

Procedure Files

A procedure file contains one or more program routines identified as a PROCEDURE or FUNCTION. A procedure file is an ASCII text file and can be created or modified with a text editor. Procedure files are included in your application automatically if identified with the SET PROCEDURE TO command unless you compile with the (-m) switch. The default extension for procedure files is ".prg".

FILE ALIASES

Specifying an Alias

An alias is an alternative name for a database. It is established when the database is opened with a USE command.

An alias is used to permit access to a field in an open database in another work area. Clipper will not permit access to a field in a database open in another work area unless the alias is specified.

There are 255 possible open databases in 255 different work areas (allowed by DOS 3.3 only; otherwise the limits are 15 open databases and 15 work areas). The SELECT command is used to move between work areas. The aliases A through L or 1 through 255 are automatically assigned to these areas, depending upon the number of open work areas.

```
USE Customers
SELECT 2
USE Invoices
```

You may also specify your own alias name when the file is opened with the USE command as in the following example.

```
USE Customers ALIAS Clients
```

Automatic Alias

Unless an alias name is specified in the USE command, Clipper assigns the database filename as the alias. This is in addition to the alias letter and number assigned.

Either the alias letter, number, the user-specified alias name or the automaticallly assigned alias name may be used to specify a field in a database open in another work area.

Fields from an open database in another work area may be accessed and specified by using the following syntax:

```
<alias>-><fieldname>
```

where <alias> is either the alias letter or number, the user specified alias name, or the automatically assigned alias name. Using aliases to access fields in other areas is usually done after execution of the SET RELATION TO command.

FIELDS

Fields are used to record and store units of information in the database file.

Fieldnames

A fieldname can contain letters, numbers, or the underscore (_) symbol. The first position in the name must be a letter. Fieldnames can have up to 10 characters and cannot contain embedded blank spaces. The various field types, identified below, are stored as part of the database file structure.

Character Fields

Character fields can contain any combination of letters, numbers, spaces, and special symbols and may be up to 32K characters in length.

Numeric Fields

Numeric fields store only numeric values and can be used in arithmetic operations. A numeric field can contain up to 19 digits with up to 18 decimal places.

You are only permitted to enter numbers, a plus sign, a minus sign, or a period (decimal point) in numeric fields.

Date Fields

Date fields store dates and always have a length of eight. Date fields are always sorted in chronological order. Clipper will not allow the entry of an invalid date.

Logical Fields

Logical fields store a single character which indicates either a logical true or a logical false. They are defined in the database by entering the fieldname and the field type "L".

Clipper will permit the entry of the following in a logical field: T or Y (upper or lower case) for true, and F or N (upper or lower case) for false. However, logical fields are stored in the database file only as the characters "T" or "F".

Memo Fields

A memo field contains text information. The text for all records is stored in a separate .dbt file. Clipper assigns a field length of 10 to the .dbt file. The length of the field for each record is determined, however, when the information is entered and can contain up to 64K characters.

CONSTANTS

A constant is a value that will be "literally" interpreted by Clipper. Constants specify actual information as opposed to information that is contained in a field or memory variable and identified by a name. In the following example, "Creditlimit" is a constant.

```
SEEK "Creditlimit"
```

Character Constants

Character constants, sometimes referred to as character strings, can contain any combination of up to 64K letters, digits, spaces, and special symbols. Character constants are identified to Clipper by enclosing them in single quotation marks (' '), double quotation marks (" "), or square brackets ([ ]). These characters are called delimiters. You must have beginning and ending delimiters and they must match. For example:

```
@ 1, 5 SAY "Enter the principal balance."
```

Any one of the three different pairs can be used to delimit a particular literal. If a second pair of delimiters is required (nested) inside the first pair, the second pair must be different than the first pair. The three kinds of delimiters permit three levels of nesting. For example:

```
? ["Welcome to L. A.", said the city's mayor.]
```

Numeric Constants

Numeric constants identify numeric values and can be used in arithmetic operations.

Numeric constants must not be delimited. If delimiters are used to enclose the number, the number will become a character constant and cannot be used in arithmetic operations (unless it is a VAL() function argument).

```
memvar = 0
```

Null Strings

Certain character operations will produce results that contain nothing. This is called a "null string."

Character memory variables containing null strings can be created by storing a string containing nothing (two consecutive quotation marks). The length of a null string is 0.

```
null = ""
```

MEMORY VARIABLES

Memory variables temporarily store information in the computer's memory. The memory variables and their values will be released when you RETURN, RELEASE ALL or CLEAR MEMORY unless you use the SAVE TO command.

Memory Variable Names

The name you assign to a memory variable (memvar) can contain letters, numbers, or the underscore symbol (_) . The first position in the name must be a letter. The memory variable name can contain up to 10 characters.

Clipper allows up to 2048 active memory variables at one time, memory permitting. They can be created by using the STORE command or by using the equal sign as an assignment operator as in the following examples:

```
STORE 0 TO price, total_cost, quantity
start_row = 5
```

Save memory variables with the SAVE TO command.

The memory variable's type is established when the variable is created. The memvar's type is the same as the value stored in it. If you store a character value for example, the memory variable created will be a character type.

A memory variable can have the same name as a database field. When the name is used, however, the database fieldname will take precedence over the memory variable. To specify the memory variable when it has the same name as a database field, use "M" as an alias.

```
M-><memvar>
```

Memory Variable List

Some commands have options which allow more than one memory variable as arguments. Separate each memory variable with a comma. You may also add a space after the comma for clarity and ease of reading. Clipper does not consider the space significant.

```
STORE 0 to sales, commission, quantity, balance
```

Scoping of Memory Variables

Memory variables are classified either PUBLIC or PRIVATE. Memory variables created in programs and procedures are PRIVATE unless declared PUBLIC.

When you declare a memory variable PUBLIC, it is available to all programs and procedures. PRIVATE memory variables can only be used in the Clipper programs or procedures where they are declared and in the lower level programs. PRIVATE memory variables are released when you RETURN to a higher level program or procedure.

Creating PRIVATE Variables

The PRIVATE command can be used in subroutines to name memory variables without concern that the routine will redefine a variable of the same name. This allows you to write programs and procedures that contain only "local" variables. Declaring a memory variable PRIVATE hides existing variables with the same name. The following program demonstrates the concept. "Mem1", which is declared PRIVATE in the called procedure, is not changed.

```
SET TALK OFF
mem1 = "one"
mem2 = "two"

DO Capitals        && Call procedure.
? mem1             && Results: "one"
? mem2             && Results: "TWO"
RETURN

PROCEDURE Capitals
PRIVATE mem1       && Hide mem1 above.
mem1 = "ONE"       && Change local mem1 only.
mem2 = "TWO"       && Change mem2 above.
? mem1             && Results: "ONE"
? mem2             && Results: "TWO"
RETURN
```

Memory Variables as Arrays

Memory variables may be declared as one-dimensional arrays, where one array equals one memory variable. The limit on the number of arrays you may declare and the number of elements they may contain is determined by available memory. The DECLARE command below is used to create an array containing 10 elements:

```
DECLARE memvar[10]
```

You may access any element of the array by including the number of the element you wish to reference in brackets.

```
FOR n = 1 TO 10
   ? memvar[n]
NEXT
```

OPERATORS

There are four different operator types used by Clipper:

Mathematical	Performs calculations on numeric expressions.
Relational	Compares two expressions and returns a logical true or false.
Logical	Compares two logical expressions and returns a logical true or false.
Character	Combines (concatenates) two or more character expressions.

Mathematical Operators

The following list shows the symbol used and the calculation performed.

Table 4-1 Mathematical Operators

Symbol	Calculation Performed
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
** or ^	Exponentiation
()	Groups sets of numbers

The modulus operator returns only the remainder of a division operation. If the remainder is desired as the result of a division, the "%" is specified in lieu of the "/" operator.

Mathematical operators may be used with date expressions, however, there are some special rules. A numeric expression may only be added to or subtracted from the date expression. The result will be the number of days added to or subtracted from the date.

Relational Operators

Below is a list of relational operators and their purpose. The two expressions being compared must be the same type.

Table 4-2 Relational Operators

Symbol	Purpose
<	Less than
>	Greater than
=	Equal to
<>, #, or !=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to
$	Is contained in the set or is a subset of
==	Compares character types for a perfect match or compares the 12 most significant digits of numeric types. Performs the same as "=" for all other types.
()	Used to change the order of operations

The dollar sign ($) operator is affected by the SET EXACT command. If SET EXACT is ON, both character expressions must be identical in content and length in order to return true.

Logical Operators

The following list shows the logical operators and the results required to return a logical true.

Table 4-3 Logical Operators

Operator	Returns true If:
.AND.	Both expressions are true
.OR.	Either expression is true
.NOT. or !	Either expression is false
$	First character expression is contained in the second character expression

In addition, parentheses can be used in the expressions to group different portions of a logical comparison.

String Operators

Both string operators concatenate character strings. The difference between them is the way in which trailing blanks contained in the expression are handled.

Table 4-4 String Operators

Operator	Purpose
+	Used to concatenate two or more character expressions. Trailing blank spaces in the expressions will be placed at the end of each expression.
-	Used to concatenate two or more character expressions. Trailing blank spaces in the expression preceding the operator will be removed from the expression and placed at the end of the expression following the minus sign operator.

Order of Evaluation for Operators

A single expression may contain a combination of several different operators and functions. Each operation is performed by the computer in a particular order; some operators take precedence over others.

When more than one type of operator appears in an expression the order of evaluation is as follows:

- String
- Mathematical
- Relational
- Logical

Expressions containing more than one operator are always evaluated from left to right. Parentheses can be used to change the precedence level of operators. If parentheses are nested, the innermost set of parentheses are evaluated first.

Each of the operator types has an evaluation order as follows:

Mathematical Operators

1. Operators contained in parentheses and functions (evaluated from left to right).
2. Exponentiation.
3. Multiplication and division (from left to right).
4. Addition and subtraction (from left to right).

In the first example below, multiplication takes precedence over addition. 3 is added to 20. In the second example, parentheses cause the addition to be performed first and the sum of 7 to be multiplied by 5.

3 + 4 * 5 = 23

(3 + 4) * 5 = 35

String Operators

1. Operators and functions contained in parentheses.
2. From the left of the expression to the right.

Relational Operators

1. Operators and functions contained in parentheses.
2. From the left of the expression to the right.

Logical Operators

1. Operators and functions contained in parentheses.
2. .NOT.
3. .AND.
4. .OR.

EXPRESSIONS

An expression consists of one or more elements that are evaluated by Clipper. These elements may be:

Fields
Memory variables
Literals (numeric and character)
Functions
Operators

The different expression types are indicated as follows:

<exp> - may be any type of expression
<expN> - numeric expressions
<expC> - character expressions
<expD> - date expressions
<expL> - logical expressions

Some commands and functions require you to enter a specified number of expressions, each meaning something different. The required information for each expression will be explained in the command and function definition. Multiple expressions are indicated by the syntax <exp1>, <exp2>, <exp3>, etc., where each expression is separated by a comma.

The following examples demonstrate each type of expression.

```
? TRIM(FIRSTNAME) + " " + LASTNAME   && Character
? creditlimit - curbalance           && Numeric
? creditdate + 90                    && Date
? curbalance > creditlimit           && Logical
```

SYNTAX RULES

In Clipper, there are rules for consistent usage of commands and functions. These rules are called "syntax."

Commands

A command consists of at least one word (called a key word). A command may require additional information which is supplied by the user. The following command, USE, opens the file Customers.dbf.

```
USE Customers
```

Commands may have additional key words and arguments which are optional.

Continuing a Command Line

Clipper allows you to continue a line within your program onto the next line by ending the first line with a semicolon. Each line may contain 256 characters and there is no practical limit on the number of characters per command.

To continue a string, end the character string on the first line with a quotation mark ("), followed by a plus sign (+), followed by a semi-colon (;). The second line must begin with a quotation mark, contain the remainder of the character string, and end with a quotation mark.

```
@ 23,0 SAY "Enter the order only if credit " + ;
   "has been approved!"
```

Note that the SET PATH TO command does not allow you to use the semicolon to continue one line onto another. See the SET PATH TO command in Chapter 5 for additional information.

Functions

Functions consist of a key word followed by left and right parentheses. They may have required or optional arguments contained in the parentheses. The following function, UPPER(), returns the character argument in upper case letters.

```
? UPPER(firstname)
```

Keywords

Capital letters are used throughout the manual to identify the key words of commands and functions. Key words should not be used as filenames, fieldnames or memory variable names.

Symbols

The following symbols are used to determine whether the requested syntax information is mandatory or optional:

[ ] Square brackets indicate an optional part of the command. Do not type in the brackets.

< > Angle brackets enclose information that you must provide, such as the name of a database, field, file or memory variable. Do not type in the brackets.

... The ellipsis indicates that there will be multiple program statements, as illustrated in the DO CASE...CASE...ENDCASE command syntax.

/ The slash is used to indicate an either/or situation in options. In this case, only one of the options can be used at a time.

Syntax Example:

```
LIST [OFF] [scope] <field list> [FOR <condition>]
[WHILE <condition>] [TO PRINT][TO FILE <filename>]
```

You provide any information in lower case, such as <field list> or <condition>. If a comma appears in the syntax, this comma should be used to separate phrases.

MACRO SUBSTITUTION

A macro allows the contents of a character memory variable to be substituted in the command line. The contents of the memory variable are evaluated literally by Clipper. The memory variable must be preceded by the ampersand sign (&).

Clipper allows the use of macros anywhere in DO WHILE statements, as well as allowing the use of nested or recursive macros. Clipper must know all of the command elements when the program is compiled so the validity of the statement can be determined and the appropriate code generated. Every line must contain a valid command and any punctuation that determines command syntax.

> **Note:** Macros cannot contain commands or any part of a command (including commas).

The following table illustrates macros which are, and are not, acceptable to Clipper.

Table 4-5 Macros in Clipper

CLIPPER WILL COMPILE	CLIPPER WILL NOT COMPILE
file = "Abc" USE &file	file = "USE Abc" &file
comp = "Nmbr2 > Nmbr1" LIST Name, City FOR &comp	comp = "Name, City FOR Nmbr2 > Nmbr1" LIST &comp

Since macros cannot contain commands, or any part of a command, the following example illustrates a method of combining a command with a macro:

```
LIST Name FOR Name = "Jones"
```

can be expressed as a command and macro such as:

```
condition = 'name = "Jones"'
LIST Name FOR &condition
```

USER-DEFINED FUNCTIONS

Clipper gives you great programming power by allowing you to create your own functions. This feature allows you to customize Clipper to suit your needs.

A representative format to be used is:

```
FUNCTION <name>
*
* Define the parameters that will be used by the
* function.
[PARAMETERS <memvar list>]
<program source code statements>
* Then insert a RETURN statement followed by a value
* that the function returns.
RETURN <value>
```

The following code defines the FUNCTION named Center and the algorithm to center a character string.

```
FUNCTION Center
*  Syntax:  Center(<expC>,<expN>)
*  Notes:  Returns the expC centered in the width
*          expN by padding leading blanks.
PRIVATE string, width
PARAMETERS string, width
IF LEN(string) >= width    && Too long to center
   RETURN (string)
ENDIF
RETURN SPACE(INT(width/2) - INT(LEN(string)/2) + ;
   string
```

User-defined functions may be placed at the bottom of your programs or in procedure files. They may then be used throughout your application as you would any other function, including index keys and the VALID clause of @...SAY...GET.

User-defined functions may also be written in C or Assembly language and linked with your application. (See the Extend System for more information and examples).

ESTABLISHING MULTIPLE RELATIONS TO A FILE

A relational data system can be described as a group of loosely related data structures which collectively contain every required piece of information on a given subject. In most cases, it is prohibitive to store all information on the subject in one location or data record. In the following example we use an invoicing file structure. Clearly it is inefficient to store entire orders in one data record. It would be impossible to know how many fields are needed for the storage of the ordered items. Further, it is inefficient to reinput a customer's information every time he orders. A relational system corrects these problems by allowing the user to store more closely related pieces of information in one location and relating those bits of data to other files as required. This is done through a code taken from each of the auxiliary files and stored in each of the detail records. In our example, an invoice number and a customer number are stored in every detail (item) record in an item data file. These codes allow access to the customer database so that addresses, terms and other credit information can be printed on the invoice while also allowing access to inventory information such as stock levels, pricing and item descriptions.

So, in its base sense a relation is a linkage from one file to another which enables two files to act as a single data source. For example, consider an invoicing program which contains the following database files:

Items.dbf - contains descriptions of inventory items, quantities on hand, etc. INDEX this file ON Item_no.

Invoice.dbf - contains the account number and shipping information for each order. INDEX this file ON Order_no.

Detail.dbf - contains the requested amounts of each inventory item for a particular order. INDEX this file ON Detail.

To set the relation for this system, enter the following commands:

```
SELECT 3
USE Items INDEX Item_no
SELECT 2
USE Invoice INDEX Order_no
SELECT 1
USE Detail INDEX Detail

* Define relations between the files
```

```
SET RELATION TO Item_no INTO Items,;
   Order_no INTO Invoice
```

Now, as you SKIP through Detail.dbf, Clipper positions the Items.dbf and Invoice.dbf files to the first record in these files matching the field from the parent file on which the relation was made.

The key expression in the parent file can be the same or different from the key expression in the related child. For example, you can relate the Items->Item_no field to the Invoice->Order_no field. Clipper finds matches based on the contents of the indexed child file, or record number if not indexed. For non-numeric fields, each child must be indexed. The information in the key expression of the parent can be a subset of the information in the key expression of the child.

DESIGNING CUSTOMIZED HELP

Context-Specific Help

When a user presses the F1 key to request the context-specific help available in your program, Clipper sends three parameters to your HELP.PRG. Pressing the F1 key is equivalent to:

```
DO Help WITH call_prg, line_num, input_var
```

The parameters are described below.

Note that Call_prg, line_num and input_var used below are sample parameter names. These names are suggestions only. You may use any legal memory variable name.

1. Call_prg - This is the name of the procedure calling HELP.PRG. It is a character variable containing the name of the procedure calling HELP.PRG. All letters in this variable will be upper case. **Any comparison made to the procedure name must be in upper case letters.**

2. Line_num - This parameter is a numeric variable containing the source code line number of the pending READ in the program calling HELP.PRG.

3. Input_var - This parameter is a character variable containing the name of the memory variable that is awaiting user input. All letters in this variable will be upper case. Wait state commands such as WAIT, INPUT and ACCEPT, however, will return null strings to the variable for this parameter. **Any comparison made to the input variable name must be in upper case letters.**

Examples:

```
input_var = 'loans' && This is incorrect

input_var = 'LOANS' && This is correct
```

Recursive calling of HELP.PRG will cause problems and should be avoided by inserting the following code at the top of HELP.PRG:

```
IF call_prg = "HELP"
   RETURN
ENDIF
```

The HELP.PRG should be written as follows:

Begin the program with a PARAMETER statement to receive the three memory variables Clipper sends to your program. The PARAMETERS statement must be present in the HELP.PRG even if parameters are not used.

> **IMPORTANT:** CLEAR and READ commands should not be used because they clear all pending GETs. Use "@ row, col CLEAR" to clear the screen. Use WAIT, ACCEPT, MENU TO or INKEY() to enter data.

Specify the appropriate instructions for each pertinent variable as in the example below:

```
* Sample Context-Specific HELP.PRG
*
* When the F1 function key is pressed while awaiting
* the input of a variable, Clipper invokes the help
* facility as though the command DO HELP WITH
* call_prg, line_num, input_var had been used.
*
* Make parameters available to
```

```
* HELP.PRG before proceeding.

PARAMETERS call_prg, line_num, input_var
*
DO CASE
    CASE call_prg = "CUSTMAST";
        .AND. line_num = 125;
        .AND. input_var = "LOANS";
        @ 23,5 SAY "Please enter interest for loantype "+;
            "&loantype."
    CASE call_prg = "CUSTMAST";
        .AND. line_num = 231;
        .AND. input_var = "ACCOUNT";
        @ 23,5 SAY "Please enter the interest "+;
            "payments for this month."
ENDCASE
RETURN
```

Generalized Help

An alternative or supplementary method for providing help for end users is the generalized help feature. This method lets you establish a set of applicable screens or provide other assistance to the user. Proceed as follows:

1. When writing the application(s), define a PUBLIC variable help_code (or some other variable name). Define the variable's value according to the section of the program you are in, as in the example below:

```
* Program CUST_SVC to demonstrate the use of
* help_code.
*
* Assign help_code a value for the first program
* section help_code = "01".
*
<program source code statements (DO, IF, CLEAR,
etc.)>
* Assign help_code value for next section of
* program help_code = "02".
<program source code statements (DO, IF, CLEAR,
etc.)>
* etc., etc.
```

2. Write the HELP.PRG to define the help to be provided for each value of help_code, that is, for each part of the program. When the F1 key is pressed by the user during program execution, your HELP.PRG is called and assistance is provided according to the instructions in the HELP.PRG. For example:

```
* Sample generalized HELP.PRG for Clipper.
*
* Assumes PUBLIC variable help_code has been
* defined in the main application.
*
* Note:  In this example, use of the commands
* SAVE SCREEN and RESTORE SCREEN eliminates the
* necessity of repainting the original input
* screen after the user has viewed the help
* information.
*
IF help_code = "01"
   SAVE SCREEN
   * Erase without clearing GETs...
   @ 0,0 CLEAR
   ? "AMORTIZATION UPDATES"
   ? "Procedures "
   ?
   ? "To see how a change in the loan terms will "
   ? "affect amortization or to actually enter "
   ? "a change."
   ?
   ? "Enter LOAN NO. to display current terms of loan."
   ?
   ? "Enter OPTION NO. indicating terms to be changed,"
   ? "then enter the new term in the adjacent space."
   ? "1 for term of loan in months - Enter new"
   ? "months."
   ? "2 for interest rate - Enter new interest rate."
   ? "3 for principal due - Enter new P.B. amount."
   ?
   ? "Press:"
   ? "F6 key to see the results of the calculation."
   ? "F7 key to make the new terms permanent."
   ? "F8 key to EXIT."
   ? "Press any key to go back to AMORTIZATION SCREEN."
   WAIT " "
   RESTORE SCREEN
   RETURN
ENDIF

IF help_code = "02"
   SAVE SCREEN
   @ 0,0 CLEAR
   ? "See page 2-343 of the Procedures manual"
   ? "or contact the Loan Service Supervisor"
   WAIT
   RESTORE SCREEN
   RETURN
ENDIF
```

DIVIDING BY ZERO

Clipper will generate a run time error whenever an attempt is made to divide by zero. This can sometimes occur when dividing by a numeric field that has not been initialized. The error message is provided to make the programmer aware of overflow conditions.

The following function will not cause an error and will return 0 if an attempt is made to divide by zero.

Example:

```
var1 = 25
var2 = 0
? ZERO(var1, var2)
* 0 will be returned
RETURN

FUNCTION Zero
PARAMETERS znum1,znum2
IF znum2 <> 0
   RETURN (znum1/znum2)
ENDIF
RETURN (0)
```

MATH PROCESSOR CHIP

An 8087, 80287 or 80387 processor chip will speed up all floating point arithmetic operations. Clipper stores numbers in databases as ASCII characters, but converts them to floating point prior to all arithmetic operations, using the math processor chip if it is present.

USE OF SCOPE IN COMMANDS

<Scope> is compatible with dBASE III PLUS. The FOR and WHILE features are not exclusive. This allows for conditional operations within a scope.

Example:

```
SEEK "CA"
LIST Balance, Company, Phone FOR Balance <> 0;
   WHILE State = 'CA'
```

dBASE III PLUS COMPATIBLE INDEXES

The default extension for dBASE III PLUS compatible indexes is ".ndx", while the default extension for Clipper indexes is ".ntx". Though they are both index files, .ndx and .ntx files are not compatible. NDX.OBJ is a linkable object file which allows you to create and use dBASE III PLUS compatible indexes with Clipper rather than the standard Clipper indexes.

Index files are created in Clipper by using the standard INDEX ON command. Clipper creates .ntx files unless you link NDX.OBJ with your compiled application to supersede the .ntx indexing scheme. Enter one of the following commands to link NDX.OBJ.

With the DOS Linker:

```
LINK MYPROG + NDX,,,\Clipper\Clipper
```

With PLINK86-Plus:

```
PLINK86 FI MYPROG, NDX LIB \CLipper\Clipper
```

These commands link MYPROG.OBJ and NDX.OBJ to create an executable file named MYPROG.EXE which uses dBASE III PLUS compatible .ndx files. Both SET ORDER TO and SET INDEX TO support the use of .ndx files within your application.

> **Note:** You cannot use both .ndx and .ntx indexes within the same application.

There are some notable differences between .ndx and .ntx files you should consider before deciding whether to use NDX.OBJ. .Ntx files are inherently faster and remain smaller than .ndx files during index manipulation.

You should also note that all applications simultaneously using an .ndx file should be of the same type; all dBASE applications or all Clipper applications. You cannot have a dBASE application using the .ndx file from one work station while a Clipper application uses it from another work station.

KEY TABLES FOR FULL SCREEN OPERATIONS

Navigation Keys

When editing a screen of GETs, the cursor may be moved as follows:

Table 4-6 Full-Screen Navigation Keys

Key	Purpose
Leftarrow, Ctrl-S	Character left. Does not move cursor to previous GET.
Rightarrow, Ctrl-D	Character right. At end of GET, cursor moves to next GET.
Ctrl-Leftarrow, Ctrl-A	Word left.
Ctrl-Rightarrow , Ctrl-F	Word right.
Uparrow, Ctrl-E	Previous GET.
Dnarrow, Ctrl-X, Return, Ctrl-M	Next GET.
Home	Beginning of GET.
End	Last character of GET.
Ctrl-Home	Beginning of first GET.
Ctrl-End	Beginning of last GET.

Editing Keys

Table 4-7 Full-Screen Editing Keys

Key	Purpose
Del, Ctrl-G	Delete character at cursor position.
Backspace, Ctrl-H	Destructive backspace.
Ctrl-T	Delete word right.
Ctrl-Y	Delete from cursor position to end of GET.
Ctrl-U	Restore current GET to original value.

Escape Keys

Table 4-8 Full-Screen Escape Keys

Key	Purpose
Ctrl-W, Ctrl-C, PgUp, PgDn	Terminate READ saving current GET.
Return, Ctrl-M	Terminate READ from last GET.
Esc	Terminate READ without saving current GET.

Mode Keys

Table 4-9 Full-Screen Mode Keys

Key	Purpose
Ins, Ctrl-V	Toggle insert mode.

5 Clipper Commands

Chapter 5 contains summarized lists and detailed descriptions of the Clipper commands. Topics covered include:

- Conventions used in command and function syntax.
- A summary of all commands used in Clipper, including the command name, syntax and purpose.
- A detailed explanation of all Clipper commands.

CONVENTIONS USED IN COMMANDS AND FUNCTIONS

Typography

This manual uses different typography to distinguish between elements of the language and the discussion of them. The following list gives the different convention for each one:

`Example`

Examples of Clipper program code are displayed in the typeface above.

KEYWORDS

Clipper commands and functions are all upper case.

Procedures

User-defined functions and procedures are capitalized, i.e., Center().

Keynames

Keynames are capitalized, i.e., Ctrl-Leftarrow.

Filenames/FILENAMES

Clipper filenames are capitalized, i.e., Myfile.prg

DOS filenames are all upper case including extension, i.e., PROG.EXE.

Clipper file types are referred to in discussion using the extension in lower case bounded by parentheses, i.e., (.txt).

Symbol definitions

The following are symbols used in syntax to describe the function of a keyword or metasymbol:

Table 5-1 Symbols Used in Syntax

Symbol	Description
<>	User input
[]	Optional
/	Exclusive or
...	Repeating elements if followed by a symbol Intervening code if followed by a keyword

Metasymbol definitions

The following are metasymbols used within syntax to describe the general nature of a command or function argument:

Table 5-2 Metasymbols Used in Syntax

Metasymbol	Description
alias	Work area name
array	Array name
condition	Logical expression
delimiter	Character constant to bound character strings
device	Print device
directory skeleton	Character expression
drive	Drive designator
exp	Expression of any type
expC	Character expression
expD	Date expression
expL	Logical expression
expN	Numeric expression
ext	File extension
field	Database field
file	Name of a file not including the extension
key exp	ExpC, expD, or expN for ordering and grouping
list	Items separated by commas
memvar	Memory variable of any type
memvarC	Memory variable of character type
parameter	Variable or expression of any data type
path	Path to specified directory from the root directory
procedure	Procedure, function, or program
process	Externally compiled or assembled procedure
program	Executable DOS program name
prompt	Character expression
row	Numeric expression for screen column
scope	Portion of database file to process consisting of a scope keyword and a expN
skeleton	Wildcard pattern including ? and *
text	A string of character constants
variable	Field or memory variable
work area	Work area number

SUMMARY OF CLIPPER COMMANDS

?/?? <exp list>

Displays the results of one or more expressions separated by a space.

@ <expN1>, <expN2>, <expN3>, <expN4> BOX <expC>

Draws a box on the screen with configurable border and fill characters.

@ <expN1>, <expN2> CLEAR [TO <expN3>, <expN4>]

Clears a rectangular region of the screen.

@ <expN1>, <expN2> PROMPT <expC> [MESSAGE <expC>]

Paints menu prompts and defines an array of messages for MENU TO.

@ <expN1>, <expN2> [SAY <exp> [PICTURE <expC>]] [GET <variable> [PICTURE <expC>] [RANGE <expN1>, <expN2>] [VALID <expL>]]

Displays and/or inputs data at specified row and column positions on either the screen or printer.

@ <expN1>, <expN2> TO <expN3>, <expN4> [DOUBLE]

Draws single or double line boxes on the screen.

ACCEPT [<prompt>] TO <memvar>

Stores a character string in a memory variable from keyboard input.

APPEND BLANK

Adds a blank record to the end of the current database file.

APPEND [<scope>] [FIELDS <field list>] FROM <file>/(<expC>) [FOR <condition>] [WHILE <condition>] [SDF]/[DELIMITED [WITH BLANK/<delimiter>/(<expC>)]]

Adds records to the current database file from an ASCII text file or another database file.

AVERAGE [<scope>] <exp list> TO <memvar list> [FOR <condition>] [WHILE <condition>]

Averages a series of numeric expressions to memory variables for a range of records in the current work area.

CALL <process> [WITH <exp list>]

Allows you to execute separately compiled or assembled routines optionally passing parameters.

CANCEL

Terminates program execution returning control to the operating system after closing all open files.

CLEAR

Clears the screen, clears all pending GETs, and homes the cursor.

CLEAR ALL

Closes all open database (and related index, format, and memo) files, releases all memory variables, and SELECTs work area 1.

CLEAR GETS

Releases pending GET variables.

CLEAR MEMORY

Releases all PUBLIC and private memory variables.

CLEAR TYPEAHEAD

Clears the keyboard buffers of any pending characters.

CLOSE ALL/ALTERNATE/DATABASES/FORMAT/INDEX

Closes the specified type of file.

COMMIT

Performs a solid-disk write for all work areas.

CONTINUE

Resumes the pending LOCATE search in the current work area.

COPY TO <file>/(<expC1>) [<scope>] [FIELDS <field list>] [FOR <condition>] [WHILE <condition>] [SDF/DELIMITED/DELIMITED WITH <delimiter>/(<expC2>)]

Copies all or part of the current database file to a new file.

COPY FILE <file1>.<ext> TO <file2>.<ext>

Duplicates a file of any kind in the current Clipper default drive and directory.

COPY STRUCTURE TO <file>/(<expC>) [FIELDS <field list>]

Creates an empty database file with field definitions from the current database file.

COPY TO <file>/(<expC>) STRUCTURE EXTENDED [FIELDS <field list>]

Creates a database file with four fields: Field_name, Field_type, Field_len, and Field_dec. The records of this new database file are the field definitions of the current database file.

COUNT [<scope>] [FOR <condition>] [WHILE <condition>] TO <memvar>

Tallies the number of records in the current work area for the specified scope and condition.

CREATE <file>/(<expC>)

Creates an empty structure extended file.

CREATE <file1>/(<expC1>) FROM <file2>/(<expC2>)

Creates a new database file from a structure extended file.

DECLARE <array1>[<expN1>] [,<array2>[<expN2>]]...

Creates one or more memory variable arrays.

DELETE [<scope>] [FOR <condition>] [WHILE <condition>]

Marks records in the current work area for deletion.

DIR [<drive>:] [<path>\] [<skeleton>]

Displays the names of the files in the specified drive and/or directory.

DISPLAY [OFF] [<scope>] <exp list> [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file>/(<expC>)]

Displays the result of one or more expressions for each record processed within the specified scope and condition.

DO <procedure> [WITH <parameter list>]

Executes the specified procedure with an optional list of actual parameters.

DO CASE...CASE...[OTHERWISE]...ENDCASE

Selects a path of program execution from a set of conditions and branches on the first true evaluation.

DO WHILE <condition>...[EXIT]...[LOOP]...ENDDO

Executes a looping structure while a condition is true (.T.).

EJECT

Advances the print head to the top of a new page and resets PROW() and PCOL() to zero.

ERASE/DELETE FILE <filename>.<ext>

Deletes the specified file from disk.

EXTERNAL <procedure list>

Declares a symbol to the linker allowing procedures placed in overlays to be referenced with macros.

FIND <character string>/(<expC>)

Positions the record pointer to the first record with an index key that matches the specified character string or expression.

FOR <memvar> = <expN1> TO <expN2> [STEP <expN3>]...[EXIT]...NEXT

Executes a loop for a range either incrementing or decrementing the range expression.

FUNCTION <procedure>...RETURN <exp>

Defines a user-defined function written in Clipper.

GO/GOTO <expN>/BOTTOM/TOP

Moves the record pointer to a specific record in the current work area.

IF...[ELSE]...ENDIF

Permits conditional execution of commands with an optional alternative.

INDEX ON <exp> TO <file>/(<expC>)

Creates a file that contains an index to records in the current database file.

INPUT [<prompt>] TO <memvar>

Allows the entry of an expression into a memory variable.

JOIN WITH <alias>/(<expC1>) TO <file>/(<expC2>) FOR <condition> [FIELDS <field list>]

Creates a new database file by merging selected records and fields from two work areas.

KEYBOARD <expC>

Stuffs the keyboard buffer with the specified string.

LABEL FORM <file1>/(<expC1>) [<scope>] [FOR <condition>] [WHILE <condition>] [SAMPLE] [TO PRINT] [TO FILE <file2>/(<expC2>)]

Displays labels from a definition held in a (.lbl) file.

LIST [OFF] [<scope>] <exp list> [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file>/(<expC>)]

Displays the result of one or more expressions for each record processed within the given scope and condition.

LOCATE [<scope>] FOR <condition> [WHILE<condition>]

Positions the record pointer to the first record matching the specified condition within the given scope.

MENU TO <memvar>

Executes a light-bar menu for the currently defined PROMPTs.

NOTE/* [<text>]/[<command>] && [<text>]

Inserts non-executing comments on a new line within a program file. Comments following a "&&" may be placed after a command and on the same line or on a line by itself.

PACK

Permanently removes records marked for deletion from the current database file.

PARAMETERS <memvar list>

Specifies memory variables to receive passed values or references.

PRIVATE <memvar list>

Hides the specified memory variables allowing you to have new memory variables of the same name in a calling program and lower level programs without overwriting values stored in the original memory variables.

PROCEDURE <procedure name>

Identifies the beginning of a procedure.

PUBLIC <memvar list>/clipper

Declares memory variables global.

QUIT

Terminates program processing, closes all open files, and returns control to the operating system.

READ [SAVE]

Enters full-screen editing mode using the current set of pending GETs.

RECALL [<scope>] [FOR <condition>] [WHILE <condition>]

Reinstates records marked for deletion.

REINDEX

Rebuilds all open index files in the current work area.

RELEASE <memvar list>/[ALL[LIKE/EXCEPT <skeleton>]]

Erases memory variables.

RENAME <file1>.<ext> TO <file2>.<ext>

Provides a new name to an existing file.

REPLACE [<scope>] [<alias>->]<field1> WITH <exp1> [,<field2> WITH <exp2>,...] [FOR <condition>] [WHILE <condition>]

Changes contents of fields to specified values.

REPORT FORM <file1>/(<expC1>) [<scope>] [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file2>/(<expC2>)] [SUMMARY] [PLAIN] [HEADING <expC3>] [NOEJECT]

Displays a tabular and optionally grouped report with page and column headings from a definition held in a (.frm) file.

RESTORE FROM <file>/(<expC>) [ADDITIVE]

Retrieves memory variables from a memory (.mem) file.

RESTORE SCREEN [FROM <memvar>]

Redisplays a screen previously displayed and stored with the SAVE SCREEN command.

RETURN [<exp>]

Terminates a procedure or program returning control to either the calling procedure or the operating system.

RUN/! <program>/(<expC>)

Executes a DOS program.

SAVE TO <file>/(<expC>) [ALL[LIKE/EXCEPT <skeleton>]]

Saves memory variables to a memory (.mem) file.

SAVE SCREEN [TO <memvar>]

Writes the current screen to a buffer or memory variable.

SEEK <exp>

Searches an index for the first key matching the specified expression.

SELECT <work area>/<alias>/(<expN>)

Switches from the current to another work area.

SET ALTERNATE TO <file>/(<expC>)

Creates a file for capturing the output from commands other than @...SAY...GET.

SET ALTERNATE on/OFF/(<expL>)

Determines whether output is echoed to the currently open alternate file.

SET BELL on/OFF/(<expL>)

Determines whether the bell rings during data entry operations.

SET CENTURY on/OFF/(<expL>)

Determines whether a date displays the century.

SET COLOR TO [<standard> [,<enhanced>] [,<border>] [,<background>] [,<unselected>]]/(<expC>)

Sets screen display attributes.

SET CONFIRM on/OFF/(<expL>)

Determines whether a Return key press is required to terminate a GET.

SET CONSOLE ON/off/(<expL>)

Determines whether screen output displays for commands other than full-screen commands and extend functions such as ACHOICE() and DBEDIT().

SET CURSOR ON/off/(<expL>)

Turns the screen cursor ON/off.

SET DATE AMERICAN/ANSI/BRITISH/ITALIAN/ FRENCH/GERMAN

Sets the format of the date type for display, function arguments, and return values.

SET DECIMALS TO <expN>

Sets the number of decimal places displayed for the results of numeric functions and calculations.

SET DEFAULT TO <drive> [:<path>]

Specifies the default drive and directory for creating and saving files.

SET DELETED on/OFF/(<expL>)

Hides/PROCESSES records marked for deletion.

SET DELIMITERS on/OFF/(<expL>)

Determines whether delimiters display for GETs.

SET DELIMITERS TO [<expC>]/[DEFAULT]

Specifies character(s) used to delimit GETs.

SET DEVICE TO SCREEN/print

Sends the results of the @ command to the SCREEN/printer.

SET ESCAPE ON/off/(<expL>)

With the default ON, pressing Esc at a GET...READ statement ignores VALID and Alt-C allows for program termination. When off, SET ESCAPE will not allow an Esc to terminate a READ and Alt-C is ignored.

SET EXACT on/OFF/(<expL>)

Requires/DOES NOT REQUIRE exact matches for character comparisons.

SET EXCLUSIVE ON/off/(<expL>)

Determines whether a database file and its associated files are opened for shared or exclusive USE.

SET FILTER TO [<condition>]

Causes a database file to appear as if it contains only records that meet the specified condition.

SET FIXED on/OFF/(<expL>)

Fixes/DOES NOT FIX the number of decimal places displayed to the current DECIMALS SETting.

SET FORMAT TO <procedure>

Activates a format that executes whenever a READ is encountered.

SET FUNCTION <expN> TO <expC>

Defines a character string to be returned when the specified function key is pressed.

SET INDEX TO [<file list>/(<expC1>),...]

Opens the specified index file(s) or closes any opened index files if no files are specified.

SET INTENSITY ON/off/(<expL>)

Sets ON/off the display of GETs in enhanced color or reverse video.

SET KEY <expN> TO [<procedure>]

Allows a procedure to be executed from any wait state when a designated key is pressed, where a wait state is any command (except INKEY()) that pauses program execution.

SET MARGIN TO <expN>

Sets the left margin of the printer and the screen.

SET MESSAGE TO <expN> [CENTER]

Sets the line where MESSAGEs associated with PROMPTs are displayed.

SET ORDER TO [<expN>]

Sets any open index file as the controlling index.

SET PATH TO [<path list>]

Specifies the search path that Clipper follows when attempting to access files.

SET PRINT on/OFF/(<expL>)

Determines whether output from commands other than @...SAY is sent to the printer.

SET PRINTER TO [<device>/<file>/(<expC>)]

Determines the destination of the printed output.

SET PROCEDURE TO [<file>]

Opens the named procedure file at compile time and includes all procedures and user-defined functions in the current object file.

SET RELATION TO [<key exp1>/<RECNO()>/<expN1> INTO <alias1>] [,TO <key exp2>/<RECNO()>/<expN2> INTO <alias2>]...

Relates work areas according to key expressions.

SET SCOREBOARD ON/off/(<expL>)

Determines whether Clipper messages appear on line zero.

SET SOFTSEEK on/OFF/(<expL>)

Allows "relative" seeking. If the record is not found, the pointer is positioned to where the record would be logically.

SET TYPEAHEAD TO <expN>

Sets the size of the keyboard buffer.

SET UNIQUE on/OFF/(<expL>)

Determines if only records with unique keys appear in an index.

SET WRAP on/OFF/(<expL>)

Allows wrapping (ribboning) in MENUs.

SKIP <expN1> [ALIAS <work area>/<alias>/(<expN2>)]

Moves the record pointer either forward or backward a specified number of records in the specified work area.

SORT [<scope>] ON <field1> [/A][/C][/D] [,<field2> [/A][/C][/D]]... TO <file>/(<expC>) [FOR <condition>] [WHILE <condition>]

Copies records within the specified scope and condition from the current work area to another database file in sorted order.

STORE <exp> TO <memvar list>/<memvar> = <exp>

Stores the result of an expression to one or more memory variables.

SUM [<scope>] <expN list> TO <memvar list> [FOR <condition>] [WHILE <condition>]

Sums a series of numeric expressions to memory variables for records within the specified scope and condition in the current work area.

TEXT [TO PRINT] [TO FILE <file>]...ENDTEXT

Displays a block of text data to the screen.

TOTAL ON <key exp> [<scope>] [FIELDS <field list>] TO <file>/(<expC>) [FOR <condition>] [WHILE <condition>]

Summarizes records by key value, summing specified numeric fields, and copying summary records to a second database file.

TYPE <file1> [TO PRINT] [TO FILE <file2>]

Displays the contents of a text file.

UNLOCK [ALL]

Releases file and record locks previously established by the lock functions (FLOCK() and RLOCK()).

UPDATE ON <key exp> FROM <alias> REPLACE <field1> WITH <exp1> [,<field2> WITH <exp2>]... [RANDOM]

Updates the current database file from another database file based on a one-to-one or one-to-many relation.

USE [<file>/(<expC1>)] [INDEX <file list>/(<expC2>) [,(<expC3>)]...] [EXCLUSIVE] [ALIAS <alias>/(<expC4>)]

Opens an existing database file (.dbf), its associated memo file (.dbt), and optionally associated index files (.ntx/.ndx) in the current work area.

WAIT [<prompt>] [TO <memvar>]

Suspends program processing until a key is pressed.

ZAP

Removes all records from the active database file.

?/??

Syntax:

?/?? <exp list>

Purpose:

To display the results of one or more expressions separated by a space.

Argument:

<exp list> is the list of values of any data type to display. The list can consist of any combination of data types including memo type.

Usage:

There are two forms of the command. ? by itself displays a carriage return/line feed before displaying the results of the expression list. The ??, by contrast, sends the output without the carriage return/line feed thus allowing successive ?? commands to display to the same line.

Note that the result of each expression in the list is separated by a space.

Examples:

This example displays on two separates lines:

```
? "Hello "
? "there"
```

Results:

```
Hello
there
```

This example, by contrast, displays on the same line:

```
? "Hello "
?? "there"
```

Results:

```
Hello there
```

Library:

CLIPPER.LIB

See also:

@...SAY, TEXT

@...BOX

Syntax:

@ <expN1>, <expN2>, <expN3>, <expN4> BOX <expC>

Purpose:

To draw a box on the screen with configurable border and fill characters.

Arguments:

<expN1> is the top row. Values may be in the range of zero to 24.

<expN2> is the left most column. Values may be in the range of zero to 79.

<expN3> is the bottom row. Values may be in the range of zero to 24.

<expN4> is the right most column. Values may be in the range of zero to 79.

<expC> is a string of eight border characters and one fill character. @...BOX draws the box using this string starting from the upper left hand corner and then proceeds clockwise.

Examples:

To draw a box covering the entire screen with graphic characters for each of the corners and the sides, and a different character to fill the box, use the following code:

```
frame = CHR(201) + CHR(205) + CHR(187) +;
        CHR(186) + CHR(188) + CHR(205) + CHR(200) +;
        CHR(186) + CHR(176)
@ 1, 0, 24, 79 BOX frame
```

The following are the border definitions for several common box types:

```
* Regular single-line box.
single = CHR(218) + CHR(196) + CHR(191) + CHR(179) +;
       CHR(217) + CHR(196) + CHR(192) + CHR(179)

* Regular double-line box.
double = CHR(201) + CHR(205) + CHR(187) + CHR(186) +;
         CHR(188) + CHR(205) + CHR(200) + CHR(186)

* Double-line top and single-line side box.
```

```
double_single = CHR(213) + CHR(205) + CHR(184) +;
                CHR(179) + CHR(190) + CHR(205) +;
                CHR(212) + CHR(179)

* Single-line top and double-line side box.
single_double = CHR(214) + CHR(196) + CHR(183) +;
              ,, CHR(186) + CHR(189) + CHR(196) +;
                 CHR(211) + CHR(186)
```

Library: CLIPPER.LIB

See also: @..CLEAR, @...TO

@...CLEAR

Syntax:

@ <expN1>, <expN2> CLEAR [TO <expN3>, <expN4>]

Purpose:

To clear a rectangular region of the screen.

Arguments:

<expN1...expN2> define the coordinates of the upper right corner. The first expression is the row and the second is the column coordinate.

Options:

To: The TO clause defines the lower right corner coordinates (<expN3> and <expN4>) of the region to CLEAR. If this clause is not specified, the corner coordinates default to row 24, column 79.

Examples:

```
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 CLEAR
```

Library:

CLIPPER.LIB

See also:

@...BOX, CLEAR, SCROLL()

@...PROMPT

Syntax:

@ <expN1>, <expN2> PROMPT <expC> [MESSAGE <expC>]

Purpose:

To paint menu prompts and define an array of messages for MENU TO.

Arguments:

<expN1> is the row where the prompt displays.

<expN2> is the column where the prompt displays.

<expC> is the prompt string.

Options:

Message: The MESSAGE clause defines the message to display each time the current PROMPT Is highlighted. The message displays on the row specified by SET MESSAGE.

Usage:

@...PROMPT is the display portion of the Clipper light-bar menu system. Each @...PROMPT statement paints a menu prompt and defines an associated MESSAGE to be displayed on the line specified by SET MESSAGE TO. The light-bar menu is then invoked with MENU TO. Prompts can be painted in any order and configuration of row and column position. MENU TO navigates them in the order they are defined.

There can be up to 32 PROMPTs per menu.

Color: PROMPTs are painted in the current standard color. The highlight appears in the current enhanced color.

Example:

```
CLEAR
SET WRAP ON
SET MESSAGE TO 23 CENTER
@ 1, 3 PROMPT "File"
@ 1, COL() + 2 PROMPT "Edit"
@ 1, COL() + 2 PROMPT "Locate"
@ 1, COL() + 2 PROMPT "Options"
@ 1, COL() + 2 PROMPT "Print"
MENU TO choice
```

Library:

CLIPPER.LIB

See also:

MENU TO, SET COLOR, SET MESSAGE, SET WRAP, ACHOICE()

@...SAY...GET

Syntax:

@ <expN1>, <expN2> [SAY <exp> [PICTURE <expC>]] [GET <variable> [PICTURE <expC>] [RANGE <expN1>, <expN2>] [VALID <expL>]]

Purpose:

To display and input data at specified row and column positions.

Arguments:

<expN1> is the row coordinate.

<expN2> is the column coordinate.

Options:

Say: The SAY clause displays the result of an <exp> of any type (including a memo field) at the specified coordinates on the current DEVICE. Clipper supports two devices: PRINT and SCREEN. If DEVICE is SET TO PRINT, output is directed to the printer. Otherwise it is directed to the SCREEN. To direct @...SAYs to a file, SET DEVICE TO PRINT and then SET PRINTER TO <output filename>. Output destined for the printer is then redirected to the specified text file.

@...SAYs to the printer behave a little differently than to the screen. If you address a printer row and column position less than the last position printed since an EJECT or SETPRC(), Clipper performs an EJECT and resets the internal PROW() and PCOL() values. Your printing logic must, therefore, proceed sequentially from left to right down the page.

SAYs display in standard color (see SET COLOR).

Get: The GET clause displays a <variable> (a field or memory variable) at a specified screen coordinate and adds it to the list of pending GETs. A subsequent READ invokes a full-screen edit mode allowing you to edit the contents of the pending GETs with a full complement of editing and navigation keys. For a complete list of keys, see READ.

Clipper supports GETting fields from other work areas if fields are referenced using the alias. For example:

```
@ 10, 10 GET alias->fieldname
```

GETs display in enhanced color unless there is an unselected COLOR SETting. If this is the case, the current GET displays in the enhanced color and all other active GETs display in the unselected color.

Note that GETs are not directed to the printer or a file with SET DEVICE TO PRINT.

Picture: The PICTURE clause defines the mask for entry into a GET and formats the output of a SAY. Clipper provides two mechanisms to control formatting: functions and templates. Functions apply to the entire SAY or GET while templates mask characters position by position.

Functions: A PICTURE function is a symbol preceded by an "@." If a template symbol follows the function, it must be preceded with a space. Note that more than one function can be applied within the same PICTURE. The following table summarizes the available functions:

Table 5-3 PICTURE Function Symbols

Func	Type	Action
A	C	Allows only alphabetic characters into a GET
B	N	Displays numbers left-justified
C	N	Displays CR after positive numbers
D	D,N	Displays dates in SET DATE format
E	D,N	Displays dates in British format, numerics in European format (comma and period reversed)
K	All	Clears GET if first key is not a cursor key
R	C	Non-template characters are inserted
S<n>	C	Allows the horizontal scrolling within a GET
X	N	Displays DB after negative numbers
Z	N	Displays zero as blanks
(	N	Encloses negative numbers in parentheses with leading spaces
)	N	Encloses negative numbers in parentheses without leading spaces
!	C	Converts alphabetic characters to upper case

Templates: Template symbols follow functions in the PICTURE string if they are specified. Each position in the output or input

stream is mapped to the symbol in the same position in the template. Clipper provides a number of template symbols as follows:

Table 5-4 PICTURE Template Symbols

Template	Action
A	Displays only alphabetic characters
N	Displays only alphabetic and numeric characters
X	Displays any character
9	Displays digits for any data type including sign for numerics.
#	Displays digits, signs, and spaces for any data type
L	Displays logicals as "T" or "F"
Y	Allows only "Y" or "N"
!	Converts an alphabetic character to upper case
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Specifies a decimal point position
,	Specifies a comma position

Other characters specified in the template overwrite the character at the same position in the source stream and output. If, however, you use the "R" function, non-template symbols specified are inserted into the display but not output if the PICTURE applies to a GET.

Range: The RANGE clause limits entry into date and numeric type variables by specifying the lower and upper bounds of acceptable input (the lower must precede the upper). If the value is not within the RANGE, an indicating message displays in the SCOREBOARD area and control returns to the GET. Note that the RANGE check is performed unless you press Esc to terminate the GET. In this case, there is no RANGE check and the <variable> is restored to its original value.

Valid: The VALID clause allows you to validate an entry into a GET with a logical expression. Like RANGE, the VALID expression evaluates whenever you attempt to terminate the associated GET unless you press Esc and ESCAPE is ON. If the expression returns false (.F.) control returns to the GET and you

cannot leave the GET until the expression returns true (.T.) or you press Esc.

Note that the expression may contain or be a user-defined function. This is useful for lookups and other types of post-processing functions. One of the unique capabilities of Clipper is that within a user-defined function called by VALID, you can change the contents of the current GET. You do this by simply STOREing or REPLACEing a new value into the current GET variable. When control returns to the GET, Clipper updates it with the new value of the variable.

Usage:

Help: You can create a help system that operates within a screen consisting of @...SAY...GETs by first SETting KEY TO a specific help procedure. Then within the procedure, use READVAR() to determine the current GET and display the appropriate help screen. Note that within a SET KEY procedure, you can change the contents of the current GET in the same way you would from within a VALID.

Examples:

The following example demonstrates the use of the VALID clause to validate input into a GET:

```
number = 0
@ 10, 10 SAY "Enter a number greater than zero:";
   GET number;
   VALID n > 0
```

The following is an example of a GET into another area:

```
SELECT 1
USE Invoice
APPEND BLANK
SELECT 2
USE Inventory
@ 1, 1 GET Invoice->Cust_No
READ
```

This is an example of the "@E" function :

```
net_income = 7125.50
@ 1,1 SAY net_income PICTURE "@E 9,999.99"
```

Result:

```
7.125,50
```

This example demonstrates "@)" function:

```
net_loss = -125.50
@ 1, 1 SAY net_loss PICTURE "@)"
```

Result:

```
(125.50)
```

This example demonstrates the "@K" function to suggest an input value, but clears it immediately if the key pressed is not a cursor key:

```
file = "Accounts"
@ 1,1 SAY "Enter file" GET file PICTURE "@K"
READ
```

This example demonstrates the "@S<n>" function which allows scrolling of the long character variable within a horizontal window:

```
long = "This is much too long."
@ 1,1 GET long PICTURE "@S10"
READ
```

Library: CLIPPER.LIB

See also: ?/??, @...TO, @...CLEAR, CLEAR, CLEAR GETS, READ, SET BELL, SET CONFIRM, SET DELIMITERS, SET DEVICE, SET FORMAT, SET INTENSITY, TEXT, COL(), ROW(), PCOL(), PROW(), SETPRC()

@...TO

Syntax: @ <expN1>, <expN2> TO <expN3>, <expN4> [DOUBLE]

Purpose: To draw single or double line boxes on the screen.

Arguments: <expN1...expN4> define the coordinates of the box. ExpN1 and expN2 define the upper left corner and expN3 and expN4 define the lower right corner. If the two row coordinates (<expN1> and <expN3>) are the same, Clipper draws a horizontal line. If the two column coordinates (<expN2> and <expN4>) are the same value, Clipper draws a vertical line.

Option: **Double:** The DOUBLE clause paints the box with a double line. If this clause is not specified, the box is painted with a single line.

Usage: @...TO is very similar to @...BOX with two exceptions. First, @...BOX allows you to define the characters of the box and second, it supports a fill character. @...TO, however, is easier to use since you do not have to remember the box characters.

Example:

```
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40 DOUBLE
```

Library: CLIPPER.LIB

See also: @...BOX, @...CLEAR, ACHOICE(), DBEDIT(), SCROLL()

ACCEPT

Syntax:

ACCEPT [<prompt>] TO <memvar>

Purpose:

To enter a string from the keyboard into a specified memory variable.

Argument:

<memvar> is the name of the memory variable where the keyboard entry is placed.

Option:

Prompt: The <prompt> is a character string displayed before the input area.

Usage:

ACCEPT takes entry from the keyboard and places it into a newly created character memory variable. Return confirms entry and is the only key that terminates ACCEPT. If Return is the only key pressed, ACCEPT creates the memory variable with a null value ("").

Example:

```
ACCEPT "Enter a value: " TO var
```

Library:

CLIPPER.LIB

See also:

@...SAY...GET, INPUT, WAIT, INKEY()

APPEND BLANK

Syntax:

APPEND BLANK

Purpose:

To add a new record to the end of the currently selected database file.

Usage:

APPEND BLANK adds a blank record and makes it the current record.

Example:

```
USE Sales
? FIELD(1)                          && Result: Salesman
? LASTREC()                         && Result: 83
APPEND BLANK
? LASTREC()                         && Result: 84
name = FIELD(1)
? EMPTY(&name)                      && Result: .T.
```

Network:

When operating under a network and the current database file is shared, APPEND BLANK attempts to add and then lock a new record. If another user has locked the database file or attempted to APPEND BLANK at the same time, NETERR() returns true (.T.). Note that a newly APPENDed record remains locked until you lock another record or perform an UNLOCK. Note also that APPENDing BLANK does not release an FLOCK() by the current user.

See Chapter 10, *Using Clipper With A Local Area Network*, for more information.

Network Example:

This example APPENDs a BLANK record if NETERR() does not return true (.T.):

```
SET EXCLUSIVE OFF
IF Net_use("Accounts", .F., 5)        && USE succeeds.
   APPEND BLANK
   IF NETERR()
      ? "Not available"
   ELSE
      <process new record>...
   ENDIF
ENDIF
```

Library:

CLIPPER.LIB

See also:

APPEND FROM, SET CARRY, SET CONFIRM, SET FORMAT

APPEND FROM

Syntax:

APPEND [<scope>] [FIELDS <field list>] FROM <file>/(<expC>) [FOR <condition>] [WHILE <condition>] [SDF]/[DELIMITED [WITH BLANK/<delimiter>/(<expC>)]]

Purpose:

To add records to the current database file from an ASCII text file or another database file.

Argument:

<file> is the name of the source file. If no type option is specified, (.dbf) is the default file type and no extension is necessary. If there is a type option specified, the file extension is assumed to be (.txt) unless specified.

Options:

Fields: If the FIELDS clause is specified, data is APPENDed only into the fields specified.

Scope: The <scope> is the portion of the source database file to APPEND FROM. In Clipper, NEXT <n> APPENDS the first <n> records and supersedes any FOR or WHILE condition. RECORD <n> APPENDs only source database file record number <n> to the target database file. The default scope is ALL records.

Condition: The FOR clause specifies the conditional set of records to APPEND FROM within the given scope. The WHILE clause specifies the set of records meeting the condition from the first record in the source file until the condition fails.

Type: There are three types of files Clipper can import: SDF, DELIMITED, and (.dbf) files.

SDF identifies a System Data Format ASCII file. Each record is a fixed length, ends with a carriage return and line feed, and the end-of-file mark is Ctrl-Z (1A hex).

DELIMITED identifies an ASCII text file, where fields are separated by commas and character fields are bounded by double quote marks (the default delimiter). Note that the delimiters are not required and Clipper correctly APPENDs character fields not bounded by them. Fields and records are variable length and end with a carriage return and line feed. The end-of-file mark is Ctrl-Z (1A hex).

DELIMITED WITH BLANK identifies an ASCII text file, where fields are separated by one space and character fields are not bounded by delimiters.

DELIMITED WITH <delimiter> identifies a delimited ASCII text file, where character fields are delimited with the specified delimiter.

Usage:

Deleted records in the source database file are APPENDed but not marked as deleted in the target database file. If DELETED is ON, however, none of the deleted source records are APPENDed.

Fields with the same names and types are APPENDed. Unlike dBASE III PLUS, however, the source and target fields must be the same data type. If they are not you will get the error message "Type conflict in REPLACE (Q/A/I)?" when you APPEND FROM.

Matching widths: If a target field is larger, Clipper pads the source data to fill it. If the target field is smaller, Clipper truncates the source data.

Example:

The following APPENDs FROM using a fields list:

```
USE Sales
APPEND FIELDS Branch, Salesman, Amount;
          FROM Branch;
          FOR  Branch = 100
```

This example demonstrates using scope to specify a record:

```
APPEND RECORD 5 FROM Temp
```

Network:

Note that APPEND FROM does not require the target file locked with FLOCK() or USEed EXCLUSIVEly in order to function properly. Clipper automatically arbitrates contention for the target database file during APPENDs.

Library:

CLIPPER.LIB

See also:

COPY, FREAD()

AVERAGE

Syntax:

AVERAGE [<scope>] <exp list> TO <memvar list> [FOR <condition>] [WHILE <condition>]

Purpose:

To average a series of numeric expressions to memory variables for a range of records in the current database file.

Arguments:

<exp list> is a list of the numeric values to AVERAGE for each record processed.

<memvar list> identifies the receiving memory variables for the averages and is created when the command executes. Existing memory variables with the same names are overwritten. This list must contain the same number of elements as the list of expressions to AVERAGE.

Options:

Scope: The <scope> is the portion of the current database file to AVERAGE. The default scope is ALL.

Condition: The FOR clause specifies the conditional set of records to AVERAGE within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Example:

```
USE Sales
AVERAGE FIELDS Amount TO avg_amount;
          FOR UPPER(Branch) = "100"
```

Library:

CLIPPER.LIB

See also:

SUM, TOTAL

CANCEL/QUIT

Syntax: CANCEL/QUIT

Purpose: To terminate program processing, close all open files, and return control to the operating system.

Usage: CANCEL or QUIT can be used from anywhere in a program system to terminate and return to the operating system. A RETURN executed at the highest level procedure performs the same action.

Library: CLIPPER.LIB

See also: QUIT, RETURN

BEGIN SEQUENCE...END

Syntax:

```
BEGIN SEQUENCE
    <statements>...
    [BREAK]
    <statements>...
END
```

Purpose:

To define a control structure for user-defined error scoping within the flow of a program.

Options:

Break: The BREAK statement branches execution to the statement immediately following the matching END statement.

Usage:

BEGIN SEQUENCE...END is a control structure that allows relatively easy definition of exception handling. When an exception occurs, issue a BREAK to branch control to the program statement immediately following the END statement that terminates the current SEQUENCE program structure. BREAK can occur in a nested or the current procedure. Nested procedures can be nested any number of levels below the BEGIN SEQUENCE structure which may also include any of the runtime error procedures. The latter allows you to define local recovery operations for runtime error conditions specific to the context in which they occur.

For more information on runtime errors, refer to Appendix F.

Examples:

This code fragment demonstrates the SEQUENCE construct within a nested or the current procedure:

```
BEGIN SEQUENCE
   <statements>...
   IF break_cond
      BREAK
   ENDIF
   <statements>...
END

<recovery statements>...
```

The following example demonstrates how you can modify the Print_error() function to BREAK to local recovery for certain types of printing errors:

```
* Main procedure.
ok = .T.
error_state = .F.
local_err = .T.
*
DO WHILE ok
    BEGIN SEQUENCE
    .
    .
    .
    END
    *
    IF error_state
        ok = Print_Recover()    && Your recovery routine.
        error_state = .F.
    ELSE
        ok = .F.
    ENDIF
    *
ENDDO

* Within Errorsys.prg.
FUNCTION Print_error
PARAMETERS name, line
PUBLIC local_err
*
IF local_err
    error_state = .T.
    BREAK
ENDIF
*
<rest of error function>...
```

Library: CLIPPER.LIB

See also: RETURN

CALL

Syntax:

CALL <process> WITH <exp list>

Purpose:

To execute separately compiled or assembled routines and programs.

Arguments:

<exp list> is the list of expressions of any data type to pass to the external process.

Usage:

CALLed programs must be defined as FAR processes ending with a FAR return. All data references consist of four-byte pointers in the form SEGMENT:OFFSET, and are on top of the stack in the order passed (see the examples below). All data types are passed by reference. Your program must preserve the BP, SS, SI, DI, ES, and DS registers.

> **Note:** Microsoft C version 5.0 places a leading underscore on function names when compiled. To call them, therefore, you must use the following form:
>
> ```
> CALL _<function>
> ```

Passing parameters: The CALL command parameter list may consist of up to seven parameters. The DX:BX and ES:BX registers point to the first parameter, similar to dBASE III PLUS. If you wish to convert a dBASE III PLUS load module, add the following statements to your .ASM file:

```
PUBLIC <proc>
```

and

```
mov ds,dx
```

Character strings are passed by reference, and are null terminated (a 0 byte at the end of the string). The length of any data item must be preserved, as the data area contains many data items consecutively in memory. If an item is lengthened, you will in all likelihood write over other data.

Numeric variables are passed as eight-byte floating point, consisting of a 53-bit characteristic and an 11-bit exponent biased by 1023. To pass numeric parameters as integers, use WORD() to convert them from the Clipper internal format to integer. If the numeric value you are passing is greater than ±32,767, it cannot be passed as an integer and therefore the use of WORD() is inappropriate.

Note also that if you use WORD() to pass a numeric value, it is passed by value.

Compiling and linking: CALLed programs must conform to the following rules:

- Processes must be in INTEL 8086 relocatable object file format with the .OBJ file extension.
- Processes must follow the C language calling and parameter passing conventions.
- Processes must be available to the linker at link time along with the library of the source compiler. You will need runtime support for any language other than assembly language. See your compiler manual for further information.

Examples:

The following two examples, the first in C and the second in assembler, change the variable "var" from "123" to "ABC."

```
var = "123"
CALL Test WITH var, "ABC"
? var
RETURN
```

To CALL a C program, use the following simple program as a basis:

```
/* Compile as large model*/
test (p1, p2)

char *p1;
char *p2;

{
    while (*p2)
    *p1++ = *p2++;
}
```

To CALL an assembly language program that accomplishes the same result, use the following example:

```
var = "123"
CALL Test WITH var, "ABC"
? var
RETURN
```

Clipper executes the CALL statement above using the following parameter passing conventions:

Stack Address	Stack	Command
sp+10	[Segment of "ABC"]	push
sp+8	[Offset of "ABC"]	push
sp+6	[Segment of var]	push
sp+4	[Offset of var]	push
sp	[Segment:offset return address]	CALL FAR TEST

Thus the assembly language routine could be as follows:
(statements with asterisks are standard and therefore required)

```
PUBLIC Test              ; * Declare process.
_PROG SEGMENT BYTE 'CODE; * Clipper code segment.

ASSUME cs:_PROG          ; * Put in Clipper segment.
;
Test   PROC   FAR        ; * Required declaration.
       push   bp         ; Stack + 2.
       movbp, sp         ; Get pointer.
       push   ds
       push   es
       cld
       lds si,[bp+10]    ; Segment offset of "ABC"
       les di,[bp+6]     ; Segment offset of "123" (var)
       mov cx,3
       rep movsb
       pop es
       pop ds
       pop bp
       ret

Test   ENDP              ; * End of process.
       _PROG ENDS        ; * End segment.
       END               ; * Program end.
```

Library: CLIPPER.LIB

See also: DO...WITH, WORD()

CLEAR

Syntax:

CLEAR [SCREEN]

Purpose:

To clear the screen, home the cursor, and clear all pending GETs.

Option:

Screen: The SCREEN clause suppresses the automatic clearing of GETs when the screen is CLEARed.

Usage:

After CLEAR erases the screen the cursor is positioned at 0,0.

If you are editing GETs, do not execute a CLEAR within a procedure invoked either by SET KEY or the VALID clause since it will release all pending GETs. To clear the screen without CLEARing GETs, use CLEAR SCREEN or @ 0, 0 CLEAR.

Library:

CLIPPER.LIB

See also:

@...CLEAR, CLEAR GETS

CLEAR ALL

Syntax: CLEAR ALL

Purpose: To close all open database (and related index, format, and memo) files, release all memory variables, and SELECT work area 1.

Library: CLIPPER.LIB

See also: CLEAR MEMORY, CLOSE, RELEASE

CLEAR GETS

Syntax: CLEAR GETS

Purpose: To release all pending GETs.

Library: CLIPPER.LIB

See also: @...CLEAR, CLEAR

CLEAR MEMORY

Syntax: CLEAR MEMORY

Purpose: To release all public and private memory variables.

Usage: CLEAR MEMORY is used when you want to release both public and private memory variables. It operates in contrast to RELEASE ALL which releases only private memory variables whose scope is the current procedure.

Example:

```
PUBLIC var
var = SPACE(10)
? TYPE("var")        && Result: C
CLEAR MEMORY
? TYPE("var")        && Result: U
```

Library: CLIPPER.LIB

See also: CLEAR ALL, RELEASE ALL

CLEAR TYPEAHEAD

Syntax:

CLEAR TYPEAHEAD

Purpose:

To empty the keyboard buffer.

Usage:

CLEAR TYPEAHEAD is useful in user-interface procedures to guarantee that keys processed from the keyboard buffer are appropriate to the current activity and not pending from a previous activity. This is particularly the case in the user functions of ACHOICE() and DBEDIT().

Library:

CLIPPER.LIB

See also:

KEYBOARD, SET TYPEAHEAD, ACHOICE(), DBEDIT(), LASTKEY(), NEXTKEY()

CLOSE

Syntax:

CLOSE ALL/ALTERNATE/DATABASES/FORMAT/INDEX

Purpose:

To CLOSE specific classes of files.

Options:

All: CLOSEs all alternate, database, and index files in all work areas. Additionally, it releases all active filters, relations, and formats.

Alternate: CLOSEs the currently open alternate file. It does not, however, release the alternate file name pointed to by SET ALTERNATE TO <filename>. SET ALTERNATE TO with no argument also CLOSEs the alternate file.

Databases: CLOSEs all open database and associated index files in all work areas and releases all active filters. It does not, however, have any effect on the active format.

Format: Releases the current format performing the same action as SET FORMAT TO with no argument.

Indexes: CLOSEs all index files open in the current work area.

Usage:

In Clipper, a number of other commands also CLOSE files including:

- QUIT
- CANCEL
- RETURN from the highest level procedure
- CLEAR ALL
- USE with no argument

Clipper also closes files from a runtime error message prompt. When you terminate program execution in response to a runtime error the following options close files.

- Quit option from (Q/A/I) prompt

- Pressing "N" from the Continue option closes all files and returns to the operating system.

Library: CLIPPER.LIB

See also: CANCEL, CLEAR ALL, CLEAR MEMORY, QUIT, RETURN, SET ALTERNATE TO, USE

COMMIT

Syntax: COMMIT

Purpose: To perform a solid-disk write for all work areas.

Usage: COMMIT flushes all Clipper buffers to DOS and then performs a solid-disk write.

Requires DOS 3.3 or greater.

Example:

```
CLEAR
USE Dbf
FOR i = 1 TO 10
   APPEND BLANK
   REPLACE Fld1 WITH i
   COMMIT            && Flush to DOS and write to disk.
NEXT
```

Library: CLIPPER.LIB

See also: SKIP

CONTINUE

Syntax:

CONTINUE

Purpose:

To resume the pending LOCATE in the current work area.

Usage:

CONTINUE searches from the current record position for the next record meeting the most recent LOCATE condition executed in the current work area. It terminates when a match is found or the end of the LOCATE scope is reached.

If the CONTINUE is successful, the matching record becomes the current record and FOUND() returns true (.T.). If it is unsuccessful, FOUND() always returns false (.F.) and the positioning of the record pointer depends on the controlling scope of the pending LOCATE.

Each work area may have an active LOCATE condition. In Clipper, a LOCATE condition remains pending until a new LOCATE condition is specified. No other commands release the condition.

Example:

```
USE Sales
? LASTREC()                          && Result: 6
LOCATE FOR Salesman = "1002"
? FOUND(), RECNO(), EOF()            && Result: .T. 3 .F.
CONTINUE
? FOUND(), RECNO(), EOF()            && Result: .T. 4 .F.
CONTINUE
? FOUND(), RECNO(), EOF()            && Result: .F. 6 .T.
```

Library:

CLIPPER.LIB

See also:

LOCATE, FOUND()

COPY

Syntax:

COPY TO <file>/(<expC1>) [<scope>] [FIELDS <field list>] [FOR <condition>] [WHILE <condition>] [SDF/DELIMITED/DELIMITED WITH <delimiter>/(<expC2>)]

Purpose:

To copy all or part of the current database file to a new file.

Arguments:

<file> is the name of the new file. If no type clause is specified, (.dbf) is the default file type and no extension is necessary. If there is a type clause specified, the file extension is assumed to be (.txt) unless specified.

Options:

Fields: The FIELDS clause specifies the list of fields to copy to the target database file. The default is all fields.

Scope: The <scope> is the portion of the current database file to COPY. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to COPY within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

SDF: The SDF clause specifies the output file type as a System Data Format ASCII file. Records are fixed length, each separated by a carriage return/line feed pair. Fields are fixed length and there is no field separator. Character fields are padded with trailing blanks, numeric fields are padded with leading blanks, date fields are written in the form "yyyymmdd," and logical fields are written in the form T/F. The end-of-file mark is Ctrl-Z (1A hex.)

Delimited: The DELIMITED clause specifies the output file type as a DELIMITED ASCII file. Records are variable length, each separated by a carriage return/line feed pair. Fields are variable length and separated by commas. Character fields are bounded by delimiters (the double quote mark is the default unless you specify a different character using the WITH clause). Leading and trailing spaces for numeric and character fields are truncated, date fields are written in the form "yyyymmdd," and

logical fields are written in the form T/F. Ctrl-Z (1A hex) is the end-of-file mark.

> **Note:** DELIMITED WITH BLANK, DIF, SYLK, and WKS file type options are not supported in addition to the TYPE keyword.

Usage:

All records contained in the active database file are copied unless limited by a scope or FOR/WHILE clause. Records marked for deletion will be copied unless DELETED is ON or a FILTER has been SET.

Examples:

This example demonstrates COPYing to another database file:

```
USE Sales
? LASTREC()                    && Result: 84
COPY TO Temp
USE Temp
? LASTREC()                    && Result: 84
```

The following examples use Testdata.dbf which has four fields, one for each data type: Char, Num, Date, Logical. For purposes of the following examples, Testdata contains the following data:

```
USE Testdata
? Char, Num, Date, Logical
```

Result:

```
Character    12.00 08/01/87 .T.
```

This example COPYs TO an SDF file:

```
COPY NEXT 1 TO Temp SDF
TYPE Temp.txt
```

Result:

```
Character    12.0019870801T
```

This example COPYs TO a DELIMITED file:

```
COPY NEXT 1 TO Temp DELIMITED
TYPE Temp.txt
```

Result:

```
"Character",12.00,19870801,T
```

This example COPYs TO a DELIMITED WITH a different delimiter:

```
COPY NEXT 1 TO Temp DELIMITED WITH '
TYPE Temp.txt
```

Result:

```
'Character',12.00,19870801,T
```

Network: When you COPY in a network environment, Clipper opens the target database file EXCLUSIVE.

Library: CLIPPER.LIB

See also: APPEND FROM, COPY FILE, COPY STRUCTURE, SET DELETED

COPY FILE

Syntax:

COPY FILE <file1>.<ext>/(<expC1>) TO <file2>.<ext>/(<expC2>)

Purpose:

To duplicate a file of any type.

Arguments:

<file1> is the name of the source file to COPY including the extension.

<skeleton2> is the name of the target file including the extension.

Usage:

COPY FILE copies files from the Clipper default drive and directory.

Example:

```
COPY FILE Test.prg TO Real.prg
? FILE("Real.prg")                && Result: .T.
```

Library:

CLIPPER.LIB

See also:

CLOSE, COPY, SET DEFAULT, USE

COPY STRUCTURE

Syntax:

COPY STRUCTURE TO <file>/(<expC>) [FIELDS <field list>]

Purpose:

To create an empty database file with field definitions from the current database file.

Arguments:

<file> is the target database file. The default extension is (.dbf) unless another is specified.

Options:

Fields: The <field list> is the set of fields to COPY to the new database structure in the order specified. The default is all fields.

Example:

```
USE Sales
? LASTREC()                          && Result: 84
*
COPY STRUCTURE FIELDS Branch, Salesman TO Temp
USE Temp
? LASTREC()                          && Result: 0
```

Library:

CLIPPER.LIB

See also:

COPY STRUCTURE EXTENDED, CREATE

COPY STRUCTURE EXTENDED

Syntax:

COPY TO <file>/(<expC>) STRUCTURE EXTENDED [FIELDS <field list>]

Purpose:

To create a database file whose contents are the field definitions of the current database file.

Arguments:

<file> is the name of the structure extended database file.

Options:

Fields: The <field list> is one or more fields in the current database file whose definitions will appear in the new database file as records in the order specified.

Usage:

COPY STRUCTURE EXTENDED creates a database file with four fields: Field_name, Field_type, Field_len, and Field_dec. The structure extended file contains the structure of the current database file with a record for the definition of each field.

Used in application programs, this permits you to create or modify the structure of a database file programmatically. To create a new database file from the structure extended file, use CREATE FROM. If you only need an empty structure extended file, use CREATE.

Examples:

```
USE Sales
COPY STRUCTURE EXTENDED TO Struc
USE Struc
LIST Field_name, Field_type, Field_len, Field_dec
```

Result:

```
1  BRANCH      C   3   0
2  SALESMAN    C   4   0
3  CUSTOMER    C   4   0
4  PRODUCT     C   25  0
5  AMOUNT      N   8   2
```

Library:

CLIPPER.LIB

See also:

CREATE, CREATE FROM, FIELD(), TYPE()

COUNT

Syntax:

COUNT [<scope>] [FOR <condition>] [WHILE <condition>] TO <memvar>

Purpose:

To tally records from the current work area to a memory variable within the specified scope and specified conditions.

Argument:

<memvar> is a memory variable assigned the COUNT result. If the variable does not exist or is not numeric type, COUNT creates it as numeric.

Options:

Scope: The <scope> is the portion of the current database file to COUNT. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to COUNT within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Example:

```
USE Sales
? LASTREC()                          && Result: 84
COUNT FOR Branch = "100" TO branch_cnt
? branch_cnt                         && Result: 4
```

Library:

CLIPPER.LIB

See also:

AVERAGE, SUM, TOTAL, LASTREC()

CREATE

Syntax:

CREATE <file>/(<expC>)

Purpose:

To create an empty structure extended database file.

Argument:

<file> is the name of the empty structure extended file.

Usage:

Like COPY STRUCTURE EXTENDED, CREATE produces a structure extended file with four fields: Field_name, Field_type, Field_len, and Field_dec and can be used in conjunction with CREATE FROM to form a new database files. Unlike COPY STRUCTURE EXTENDED, CREATE produces an empty database file and does not require the presence of another database file to create it.

Library:

CLIPPER.LIB

See also:

CREATE FROM, COPY STRUCTURE EXTENDED

CREATE FROM

Syntax:

CREATE <file1>/(<expC1>) FROM <file2>/(<expC2>)

Purpose:

To create a new database file from an structure extended file.

Arguments:

<file1> is the name of the new database file to create from the structure extended file (<file2>). The field definitions in the new database file are taken from the records of <file2>.

<file2> is the name of a structure extended file to use as the structure definition for the new database file.

Usage:

CREATE FROM produces a new database file with the field definitions taken from the contents of a structure extended file. To qualify as a structure extended file, a database file must contain the following four fields:

Table 5-5 Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	3	0

Data dictionaries: For data dictionary applications, you can have any number of other fields within the structure extended file to describe the extended field attributes. You may, for example, want to have fields to describe such field attributes as a description, key flag, label, color, picture, and a validation expression for the VALID clause. When you CREATE FROM, Clipper creates the new database file from the required fields only. All other fields in the extended structure are ignored. Moreover, Clipper is not sensitive to the order of the required fields.

Field lengths greater than 999: It is possible to create a character field greater than 999 characters by specifying the Field_dec equal to the INT() of the desired length divided by 256

and the Field_len equal to the remainder of the length divided by 256.

Examples:

```
CREATE New FROM Struc
USE New
? LASTREC()                    && Result: 0
```

This example is a procedure that simulates an interactive CREATE utility:

```
CREATE New_stru
USE New_stru
more_flds = .T.

* Get the new database field names.
DO WHILE more_flds
   APPEND BLANK
   @ 5, 0 SAY "Field name     : " GET Field_name
   @ 6, 0 SAY "Field type     : " GET Field_type
   @ 7, 0 SAY "Field length   : " GET Field_len
   @ 8, 0 SAY "Field decimals : " GET Field_dec
   READ
   more_flds = (.NOT. EMPTY(Field_name))
ENDDO

* Remove all blank records.
DELETE ALL FOR EMPTY(Field_name)
PACK
USE

* Create the new database file.
CREATE Newfile FROM New_stru
ERASE  New_stru.dbf
```

To create a field with 4000 characters:

- Field_dec = INT(4000/256) = 15
- ld_len = 4000 % 256 = 160

```
REPLACE Field_name WITH "Note" ;
   Field_type WITH "C",;
   Field_len  WITH 160,;
   Field_dec  WITH 15
```

Library: CLIPPER.LIB

See also: COPY STRUCTURE EXTENDED, CREATE

DECLARE

Syntax:

DECLARE <array1>[<expN1>] [,<array2>[<expN2>]]...

Purpose:

To create one or more memory variable arrays.

Arguments:

<array> is the name of an array to create. Note that you can create more than one array with a single DECLARE statement.

<expN> is the number of elements in the array up to a maximum of 4096. An array DECLAREd with less than one element defaults to one; more than 4096 elements defaults to 4096.

> **Note:** The square brackets surrounding <expN> are a required part of the command syntax and in this case do not signify an optional argument.

Usage:

DECLARE creates single dimensional private arrays whose scope is the current procedure. At the same time, PUBLIC and private arrays created in higher-level procedures with the same name are hidden. An array uses one memory variable slot of the allotted memory variables.

Unlike memory variables, arrays and array elements cannot be saved in (.mem) files.

To assign a value to an array element, use the assignment operator (=) or the STORE command. To STORE a value to an entire array, use AFILL(). To retrieve a value from an array, refer to the element using a subscript indicating its position in the array.

To determine the number of elements in an array, use LEN() by specifying only the array name as the function argument.

Data types: Elements within the same array can be mixed type and obey all the typing rules of ordinary memory variables. TYPE() returns an "A" for an array reference and the data type of an element if the reference includes the subscript.

Use within macro variables: References to arrays and array elements can be made with macro variables with one exception: the brackets cannot be in a macro variable when DECLAREing the array. Note, however, that you cannot use or expand an array element as a macro variable. If you wish to use an array element in a macro, assign the contents of the array element to a memory variable and then use the memory variable as the macro variable. For example:

```
DECLARE fields[FLDCOUNT()]
AFIELDS(fields)
fld_name = fields[1]
? &fld_name.
```

Passing parameters: Array and array elements can be passed as parameters to Clipper procedures, user-defined functions, and external procedures that use the Extend System (see Chapter 11, *The Extend System*). Arrays are passed by reference and array elements are passed by value. You cannot, however, pass entire arrays to external procedures using the CALL command. In this case, you can only pass array elements one at a time.

PUBLIC arrays: To create PUBLIC arrays, use the PUBLIC statement. Note that PUBLIC arrays follow all the same rules as private arrays with the exception of the scope which is all procedures. See PUBLIC for more information.

Examples:

The following example creates an array, assigns values to elements, and then displays them:

```
DECLARE array[5]
array[1] = "Hello"
array[2] = 1234
? array[1]                && Result: Hello
? array[2]                && Result: 1234
```

This example demonstrates how to refer to arrays with macro variables:

```
name = "array"
number = 5
name_len  = "array[1]"
*
* Legal use.
DECLARE &name.[number]  && Creates "array" with 5
```

```
                                && elements.
&name_len. = 100                && Assigns 100 to element 1.
&name.[3]  = "abc"              && Assigns "abc" to element 3.
*
* Illegal use.
DECLARE &name_len.              && The brackets cannot be
                                && within the macro variable.
```

Library: CLIPPER.LIB

See also: PRIVATE, PUBLIC, ADEL(), ADIR(), AFILL(), AINS(), ASCAN(), ASORT()

Passing parameters: Array and array elements can be passed as a parameters to Clipper procedures, user-defined functions, and external procedures that use the Extend System (see Chapter 11, *The Extend System*). Arrays are passed by reference and array elements are passed by value. You cannot, however, pass entire arrays to external procedures using the CALL command. In this case, you can only pass array elements one at a time and by value.

Examples:

The following example creates an array, assigns values to elements, and then displays them:

```
DECLARE array[5]
array[1] = "HELLO"
array[2] = 1234
? array[1]                && Result: HELLO
```


Repeating the Error Checking Process

Unfortunately, you do not have the option of saving the perfected machine-language code that was created in the previous run, or turning off the line by line error-checking and execution process. Each time that you run the program, every line must be scrutinized by the interpreter, which significantly increases the time required to run the program.

Overview Of A Compiler

When using a compiler, the compiler checks each line for command syntax, etc., in much the same way as an interpreter. However, as opposed to the interpreter, the compiler displays the line number and associated error message for each error encountered, until the end of the program is reached.

The compiler accomplishes an intermediate step of converting the program source code into object code and saves the accumulated lines in an object code (.OBJ) file. This object file permanently saves the analyzed and converted source code.

The object code cannot be executed until it is "linked", which pulls in all supporting "run-time" routines to the object code to create a stand-alone executable (.EXE) file. After the program has been linked successfully, the .EXE file contains everything that you need to run your program directly from DOS, by simply typing in the program name. The program is run at maximum speed, as there is no longer a need to check each line of code as with an interpreter.

Using Other Languages

With many compilers, you may also link separately compiled or assembled routines into your program. The advantage of this is that these routines can be written in a different compiled

DELETE

Syntax:

DELETE [<scope>] [FOR <condition>] [WHILE <condition>]

Purpose:

To mark records for deletion in the currently selected database file.

Options:

Scope: The <scope> is the portion of the current database file to DELETE. If a scope is not specified, DELETE acts only on the current record. If a condition is specified the default scope becomes ALL.

Condition: The FOR clause specifies the conditional set of records to DELETE within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Usage:

DELETE tags records so they can be filtered with SET DELETED ON, queried with DELETED(), and physically removed from the database file with PACK. In addition, display commands such as LIST and DISPLAY identify deleted records with an asterisk (*). Once records are deleted, you can reinstate them by using RECALL. If you need to remove all records from a database file, use ZAP instead of DELETE ALL and PACK.

Example:

```
USE Sales
? RECNO()                    && Result: 1
DELETE
? DELETED()                  && Result: .T.
```

Network:

In a network environment, DELETE requires the current record be locked with RLOCK() if you are DELETEing a single record. If you are DELETEing several records the current database file must be locked with FLOCK() or USEed EXCLUSIVEly.

Library:

CLIPPER.LIB

See also:

PACK, RECALL, SET DELETED, ZAP, DELETED(), EMPTY()

DIR

Syntax:

DIR [<drive>:] [<path>\] [<skeleton>]

Purpose:

To display a listing of files from the specified path.

Option:

Skeleton: The <skeleton> is a standard wildcard (* and ?) notation for files. If specified, DIR displays all matching files. If not specified, DIR displays a list of database files.

Usage:

DIR displays two formats of listing depending on whether you specify a <skeleton>. Not specifying a skeleton displays a standard listing of database files from the current directory and includes the database filename, date of last update, and number of records. Specifying a skeleton displays a list of files including filename, extension, number of bytes, and date of last update.

To create applications where you present directory information to the user, use ADIR() instead of DIR. ADIR() allows you to fill a number of arrays with the attributes of files. Then with these arrays you can create any kind of file selection or display menu.

Library:

CLIPPER.LIB

See also:

ADIR()

DISPLAY

Syntax:

DISPLAY [OFF] [<scope>] <exp list> [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file>/(<expC>)]

Purpose:

To display the result of one or more expressions for each record processed.

Argument:

<exp list> is the list of values displayed for each record processed.

Options:

Scope: The <scope> is the portion of the current database file to DISPLAY. The default is the current record. If a condition is specified the scope becomes ALL.

Condition: The FOR clause specifies the conditional set of records to DISPLAY within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Off: The OFF clause suppresses the display of the record number.

Print: The TO PRINT clause echoes output to the printer.

File: The TO FILE clause echoes output to the specified filename. If an extension is not specified, (.txt) is added.

Usage:

DISPLAY sends the results of the <exp list> to the screen in a tabular format, each column separated by a space. Unlike other dialects, however, it does not display column headers or pause every 15 records. In fact, DISPLAY is identical to LIST with the exception that its default scope is NEXT 1 rather than ALL.

Examples:

```
USE Sales
DISPLAY DATE(), TIME(), Branch
DISPLAY Branch, Salesman FOR Amount > 500;
   TO PRINT
```

To interrupt a DISPLAY, use INKEY() as a part of the condition as follows:

```
USE Sales
DISPLAY Branch, Salesman, Amount FOR INKEY() <> 27
```

Library: CLIPPER.LIB

See also: LIST

DO

Syntax:

DO <procedure> [WITH <parameter list>]

Purpose:

To execute a procedure.

Arguments:

<procedure> is the name of the procedure to execute. This can be a procedure written in Clipper, C, or assembly language.

Option:

With: Specifies a <parameter list> of up to 128 items allowing you to pass memory variables or the results of expressions as parameters. Memory variables can either be passed by reference or value.

Usage:

Passing parameters: Using the WITH clause passes parameters to the specified procedure evaluating each expression before branching. To optionally capture the passed parameters in a Clipper procedure, there must be a PARAMETERS statement followed by the list of variables to receive the passed references or values. If the passed parameter is an expression, field, or memory variable bounded by parentheses, it is evaluated and its value passed to the called procedure. If the parameter is a memory variable not bounded by parentheses, it is passed by reference and any changes to the receiving variable are reflected in the source variable when the calling procedure terminates. Note that fields can only be passed by value and not by reference.

See PARAMETERS for a more detailed discussion of passing parameters.

Compiling: When Clipper encounters a DO statement and the specified procedure is not already known, it searches the current directory for a (.prg) file with the same name and compiles it if it is found. Otherwise the called procedure is assumed to be an external. Note that when you link, the linker will expect to resolve this external reference.

For more information on compiling and linking, refer to Chapter 7.

Examples:

```
DO AcctsRpt
DO QtrRpt WITH "2nd Quarter Report", "Sales Division"
*
number = 12
DO YearRpt WITH number, + 12, (number)
```

Library:

CLIPPER.LIB

See also:

PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, RETURN, SET PROCEDURE

DO CASE

Syntax:

```
DO CASE
CASE <condition>
    <commands>...
[CASE <condition>]
    <commands>...
[OTHERWISE]
    <commands>...
ENDCASE
```

Purpose:

To select a path of program execution from a set of conditions and branching on the first true evaluation.

Options:

Otherwise: If none of the preceding CASE conditions are true, the next set of commands following the OTHERWISE statement are executed up to the next ENDCASE statement.

Usage:

The DO CASE structure works by branching execution to the commands following the first true evaluation of a CASE condition. Execution continues until the next CASE, OTHERWISE, or an ENDCASE is encountered. At this point, control branches to the first command line following the ENDCASE.

No nesting limits: Any number of commands including other structures (DO WHILE, DO CASE, IF, FOR) may be nested within a single DO CASE structure. In addition, there is no fixed limit on the number of CASE statements that can be contained within a single DO CASE structure.

Note that like all other control structures, only the first three characters of the END statement are significant.

Examples:

```
* Display menu choices.
CLEAR
@ 1, 25 SAY    "Main Menu"
@ 3, 25 PROMPT "1. First choice"
@ 4, 25 PROMPT "2. Second choice"
@ 5, 25 PROMPT "Quit"

* Get menu key.
MENU TO choice
```

```
* Perform an action based on your menu choice.
DO CASE
CASE (choice % 3) = 0
     RETURN
CASE choice = 1
     DO One
CASE choice = 2
     DO Two
ENDCASE
RETURN
```

Library: CLIPPER.LIB

See also: DO, DO WHILE, IF, IF()/IIF()

DO WHILE

Syntax:

```
DO WHILE <condition>
    <commands>...
    [EXIT]
    <commands>...
    [LOOP]
    <commands>...
ENDDO
```

Purpose:

To execute a looping structure while a condition is true (.T.).

Argument:

<condition> is the controlling condition evaluated whenever the DO WHILE statement executes.

Options:

Exit: The EXIT statement unconditionally branches control from within a DO WHILE structure to the statement immediately following the ENDDO.

Loop: The LOOP statement branches control to the last executed DO WHILE command line.

Usage:

The DO WHILE control structure executes a block of statements repetitively as long as the specified condition evaluates to true (.T.). When the condition evaluates to true (.T.), control passes into the structure and proceeds until an EXIT, LOOP, or ENDDO is encountered. ENDDO returns control to the DO WHILE statement and the process repeats itself.

If it evaluates to false (.F.), the DO WHILE construct terminates and control passes to the statement immediately following the ENDDO.

Exiting: EXIT is used generally when you want to terminate a DO WHILE structure based on an intermediate and not the DO WHILE condition.

Looping: LOOP is generally used where you want to prevent execution of statements within a DO WHILE based on an intermediate condition and branch immediately back to the DO WHILE command line. For example:

```
DO WHILE <condition>
   <initial processing>...
   IF <intermediate condition>
      LOOP
   ENDIF
   <continued processing>...
ENDDO
```

Repeat until: the following demonstrates the use of DO WHILE to create a repeat until looping construct:

```
more = .T.
DO WHILE more
   <statements>...
   more = (<condition>)
ENDDO
```

Traversing a database file: the following two examples demonstrate the looping construct to move sequentially through a database file.

```
DO WHILE .NOT. EOF()
   <statements>...
   SKIP
ENDDO
```

This example sequentially scans a database file processing records that match a condition.

```
LOCATE FOR <condition>
DO WHILE FOUND()
   <statements>...
   CONTINUE
ENDDO
```

Macros on the DO WHILE command line: Macro variables can comprise all or part of the DO WHILE condition without limitation.

Note that like all other control structures, only the first three characters of the END statement are significant.

Library: CLIPPER.LIB

See also: FOR, IF, LIST, RETURN

EJECT

Syntax:

EJECT

Purpose:

To advance the print head to the top of a new page.

Usage:

EJECT causes the print head to advance to the top of a new page by sending a form-feed character (ASCII 12) to the printer. In addition, EJECT sets the internal printer row and column values to zero.

If you need to reset the internal printer row and column values to zero without sending a form-feed, use SETPRC().

Library:

CLIPPER.LIB

See also:

SETPRC()

ERASE/DELETE FILE

Syntax: ERASE/DELETE FILE <file>.<ext>

Purpose: To remove a file from disk.

Argument: <file> is the name of the file, including extension, to be deleted from disk.

Example:

```
? FILE("Temp.dbf")          && Result: .T.
ERASE Temp.dbf
? FILE("Temp.dbf")          && Result: .F.
```

Library: CLIPPER.LIB

See also: CLOSE, USE, FILE()

> **Warning:** Files must be CLOSEd before ERASEing them.

EXTERNAL

Syntax:

EXTERNAL <procedure list>

Purpose:

To declare symbol(s) for the linker.

Argument:

<procedure list> is the list of procedures, user-defined functions, and format files to add to the symbol table.

Usage:

Procedures, user-defined functions, and SET KEY procedures must be declared EXTERNAL if they are called with a macro or placed in overlays.

Examples:

```
EXTERNAL p1, p2, p3
routine = "1"
DO P&routine        && P1 can be in an overlay.
```

Library:

CLIPPER.LIB

FIND

Syntax:

FIND <character string>/(<expC>)

Purpose:

To search an index for the first key matching the specified character string and position the record pointer to the corresponding record.

Argument:

<character string> is all or part of the index key of a record you are searching for. If an expression (<expC>) is specified instead of a literal character string, FIND operates the same as SEEK.

Usage:

FIND searches the controlling index starting with the first key and proceeds until a match is found or there is a key value greater than the search argument. If there is a match, the record pointer is positioned to the record number found in the index. If SOFTSEEK is OFF (the default) and FIND does not find a record, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). If SOFTSEEK is ON, the record pointer is positioned to the record with the first key value greater than the search argument and FOUND() returns false (.F.). In this case, EOF() only returns true (.T.) if there are no keys in the index greater than the search argument.

Leading blanks: If a literal search argument has leading blanks, it must be delimited with quote marks and have the same number of leading blanks as the index key.

Macro substitution: The search argument can be a macro variable if not bounded by parentheses. The syntax for this is &<memvar>.

Numeric keys: Clipper stores numeric index keys with leading zeros. This means that to FIND a numeric key with a literal search argument, you must pad it with leading zeros so there are as many digits in the search argument as there are whole number digits in the index key. The number of decimal digits does not matter unless the number contains decimal values other than zero.

Examples:

```
USE Sales INDEX Branch
? LASTREC()                          && Result: 84
```

```
? INDEXKEY()                          && Result: BRANCH

FIND 200
? FOUND(), EOF(), RECNO()             && Result: .T. .F. 5

FIND 500
? FOUND(), EOF(), RECNO()             && Result: .F. .T. 85

string = "200"
FIND &string
? FOUND(), EOF(), RECNO()             && Result: .T. .F. 5

FIND "100"
? FOUND(), EOF(), RECNO()             && Result: .T. .F. 1
```

Library: CLIPPER.LIB

See also: INDEX, LOCATE, SEEK, SET INDEX, SET ORDER, SET SOFTSEEK, EOF(), FOUND()

FOR...NEXT

Syntax:

FOR <memvar> = <expN1> TO <expN2> [STEP <expN3>]
    <commands>...
    [EXIT]
    <commands>...
NEXT

Purpose:

To loop for a range either incrementing or decrementing the range expression.

Arguments:

<memvar> is the loop control variable.

<expN1> is the initial value assigned to the control variable and the lower boundary of the looping range.

<expN2> is the upper boundary of the looping range.

Options:

Step: The STEP clause sets the increment of the control variable to <expN3>. If no STEP clause is specified, the default increment is one.

Exit: The EXIT clause unconditionally branches control from within the FOR...NEXT control structure to the statement immediately following the NEXT statement.

Usage:

FOR.. .NEXT allows you to loop from an initial value of a control variable to some upper boundary moving through the range of values of the control variable for a specified increment. Each time the FOR statement executes, Clipper evaluates all command line expressions. This means that the upper boundary and increment are dynamic and can change as the construct operates.

Examples:

The FOR...NEXT construct is most useful for traversing arrays. For example:

```
* To walk forward through an entire array.
len_array = LEN(array)
FOR i = 1 TO len_array
   <commands>...
NEXT
```

```
* To walk backwards through an entire array.
len_array = LEN(array)
FOR i = len_array TO 1 STEP -1
   <statements...>
NEXT
```

Library: CLIPPER.LIB

See also: DO CASE, DO WHILE, IF

FUNCTION

Syntax:

FUNCTION <procedure>...RETURN <exp>

Purpose:

To declare a user-defined function written in Clipper.

Arguments:

<procedure> is the declared name of the user-defined function. Procedure and user-defined function names can be up to 10 characters in length.

<exp> is the function return value. All user-defined functions must return a value.

Usage:

User-defined functions are the same as procedures with two exceptions. They must begin with the FUNCTION declaration and contain a RETURN statement with an argument (in order to return a value).

To call a user-defined function, use the same notation as you would when calling Clipper functions:

```
function(<parameter list>)
```

If you are not concerned with the return value, you can use the same notation to place a user-defined function on a line by itself. In this case the return value is ignored.

Passing parameters: Parameters passed to user-defined functions are passed by value with two exceptions. First, if the actual parameter is an array reference, the entire array is passed by reference. Second, if the actual parameter is preceded by the "at" sign (@), it is passed by reference.

For a more detailed discussion on passing parameters, refer to the entry for PARAMETERS in this chapter.

Examples:

The following example uses a user-defined function to center a string within an @...SAY statement:

```
@ 12, Center("Hi there") SAY "Hi there"
RETURN

FUNCTION Center
```

```
PARAMETERS string
RETURN INT((80 - LEN(string)) / 2)
```

In this example, a variable is passed to a user-defined function by reference preceding the user-defined function argument (actual parameter) with an "at" sign (@). Note how changes to the formal parameter in the user-defined function change the original variable.

```
value = 10
? Changvar(value)
? value                                && Result: 10
? Changvar(@value)
? value                                && Result: 20
RETURN

FUNCTION Changvar
PARAMETER var
var = var * 2
RETURN (var)
```

This example demonstrates the use of a user-defined function in a VALID clause in order to validate data entry:

```
num = 0
@ 1, 0 SAY "Enter number: " GET num VALID Valnum(num)
READ
RETURN

FUNCTION Valnum
PARAMETER number
RETURN (number > 10 .AND. number < 20)
```

Library: CLIPPER.LIB

See also: PROCEDURE, PARAMETERS, RETURN

GO/GOTO

Syntax:

GO/GOTO <expN>/BOTTOM/TOP

Purpose:

To move the record pointer to a specific record in the current work area.

Argument:

<expN> is the specific record to move the record pointer to. GOTO moves the record pointer to this record even if DELETED is ON or it falls outside the scope of the current filter.

Options:

Bottom: GO/GOTO BOTTOM moves to the last logical record in the current work area if there is an active index or LASTREC() if there isn't one. If DELETED is ON or an active FILTER, this is the last record in the filter scope.

Top: GO/GOTO TOP moves to the first logical record in the current work area if there is an active index or record 1 if there is no index in USE. If DELETED is ON or there is an active FILTER, this is the first record in the filter scope.

Examples:

```
USE Sales
? LASTREC()                    && Result: 84
GO TOP
? RECNO()                      && Result: 1
GO BOTTOM
? RECNO()                      && Result: 84
GO 5
? RECNO()                      && Result: 5
GO 5 + 15
? RECNO()                      && Result: 20
```

Library:

CLIPPER.LIB

See also:

SKIP, LASTREC(), RECNO()

IF

Syntax:

```
IF <condition>
    <statements>...
[ELSEIF <condition>]
    <statements>...
[ELSE]
    <statements>...
ENDIF
```

Purpose:

To select a path of program execution from one or more conditions and branch on the first true evaluation.

Argument:

<condition> is a control expression. If it evaluates to true (.T.), all following commands are executed until an ELSEIF, ELSE, or ENDIF is encountered.

Options:

Elseif: The ELSEIF clause identifies commands to be executed when the <condition> evaluates to true (.T.). You can specify any number of ELSEIF statements within the same IF...ENDIF control structure.

Else: The ELSE clause identifies commands to be executed when the <condition> evaluates to false (.F.).

Usage:

The IF control structure works by branching execution to statements following the first true (.T.) evaluation of the IF or any ELSEIF condition. Execution then continues until the next ELSEIF, ELSE, or ENDIF is encountered whereupon execution branches to the first statement following the ENDIF. If no condition evaluates to true (.T.), control passes to the first statement following the ELSE statement.

Note that IF...ENDIF structures may be nested within other IF...ENDIF structures and other structured programming commands. These structures, however, must be properly nested.

Note also that like all Clipper control structures, only the first three characters of the END statement are significant.

Example:

```
number = 0
IF number < 50
   ? "Less than 50"
ELSEIF number = 50
   ? "Is equal to 50"
ELSE
   ? "Greater than 50"
ENDIF
```

Library: CLIPPER.LIB

See also: DO CASE, IF()/IIF()

IF

Syntax:

IF <condition>
    <commands>...
[ELSE]
    <commands>...
ENDIF

Purpose:

To conditionally execute a block of commands.

Argument:

<condition> is the control expression. If it evaluates to true (.T.), all following commands are executed until an ELSE or ENDIF are encountered. If the condition evaluates to false (.F.), control passes to the first command after the ELSE statement, if there is one. Otherwise control passes to the program statement immediately following the ENDIF.

Options:

Else: The ELSE clause identifies commands to be executed when the <condition> evaluates to false (.F.).

Usage:

IF...ENDIF structures may be nested within other IF...ENDIF structures, and other structured programming commands. These structures must be properly nested.

Note that like all the control structures, only the first three characters of the END statement are significant.

Example:

```
number = 0
IF number <= 50
   ? "Less than or equal to 50"
ELSE
   ? "Greater than 50"
ENDIF
```

Library:

CLIPPER.LIB

See also:

DO CASE, IF()/IIF()

INDEX

Syntax:

INDEX ON <key exp> TO <file>/(<expC>)

Purpose:

To create a file that contains an index to records in the current database file.

Arguments:

<key exp> is an expression that returns the key value to place in the index for each record in the current database file. The maximum length of the index key expression is 250 characters.

<file> is the name of the index file to create. Normally, the file extension is (.ntx). If, however, you have linked NDX.OBJ in order to use dBASE III PLUS compatible index files, the extension is (.ndx).

Usage:

When an index file is used, the database records appear in key expression order although the index does not alter the physical order of records in the database file. This allows you to create and maintain many logical orders of records automatically.

Clipper uses a variation of the b-tree indexing scheme with an index page size of 1024 bytes. The (.ntx) has a header for the index to tell the system what the index key is, where the "top" of the tree is, etc. In addition, Clipper keeps a pointer to the place in the file where the most recently discarded page is kept. As you modify keys in an index, the original key is removed from its location and the new key is inserted elsewhere. There are times when this will cause a page to become empty. When this happens, Clipper saves the position in its header so that it can re-use this space when it needs to create a new page later on.

Deleted and filtered records: Records that are filtered or marked for deletion are included in the index.

Date indexes: Clipper supports date indexes for both (.ntx) and (.ndx) index types. For a key expression that includes a date as a subset of the key, create the expression as character type and use DTOS() to convert the date to character. For example:

```
USE Invoices
INDEX ON Customer + DTOS(Inv_date) TO Invoice
```

Descending order indexes: To create descending order indexes or descending suborders, use DESCEND(). This function accepts any data type as its argument and returns the inverse. For example, this code fragment creates an index of invoices in descending chronological order:

```
USE Invoices
INDEX ON DESCEND(Inv_date) TO Inv_stack
```

This example, by contrast, creates an index of invoices by Customer in descending chronological order:

```
INDEX ON Customer + DESCEND(DTOS(Inv_date));
   TO Inv_stack
```

Be sure to use DESCEND() as a part of the SEEK expression when doing subsequent lookups.

Compatible index files: Clipper supports dBASE III PLUS compatible index files by linking NDX.OBJ. See Chapter 4, *The Clipper Language* for more information on compatible indexes.

Unique indexes: When you INDEX with UNIQUE ON, Clipper creates an index with uniqueness as an attribute. As indexing proceeds and two or more records have the same key value, Clipper includes only the first record in the index. Whenever the unique index is updated, REINDEXed, or PACKed, only unique records are added. This happens without regard to the current UNIQUE SETting.

Note that this differs from previous versions of Clipper where UNIQUE was a global SETting and applied to the creation and updating of all open indexes.

TRIM() in key expressions: Index key sizes under Clipper are calculated by evaluating the key expression on a blank record. The TRIM() of any part of a key expression that includes a field, therefore, always evaluates to a null string. This can lead to a size mismatch between the target and the defined key length. You can, however, build an index key on the TRIM() of a field as long as you pad the key with the number of spaces equal to the length of all the trimmed fields. For example, suppose you have two fields, Last and First, each 20 characters in length. You want to INDEX on the following expression:

```
TRIM(Last) + First
```

The actual expression you INDEX ON is this:

```
SUBSTR(TRIM(Last) + First + SPACE(20), 1, 40)
```

Note that using TRIM() to save space in the index file does not work. Clipper allocates space for keys in fixed increments and using TRIM() only confuses the issue. To create smaller index files, create smaller fixed length keys using SUBSTR() instead.

Examples:

```
? TYPE("Branch")                    && Result: C
INDEX ON Branch TO Branch
*
? TYPE("Amount")                    && Result: N
INDEX ON Amount TO Amount
*
? TYPE("Date")                      && Result: D
INDEX ON Date TO Date
```

Library:

CLIPPER.LIB

See also:

CLOSE, FIND, REINDEX, SEEK, SET INDEX, SET ORDER, SET UNIQUE, USE, DTOS(), INDEXEXT(), INDEXKEY(), INDEXORD()

INPUT

Syntax:

INPUT [<prompt>] TO <memvar>

Purpose:

To enter an expression from the keyboard and place the result in a specified memory variable.

Arguments:

<memvar> is the name of the memory variable where the result of the evaluation is placed.

Options:

Prompt: The <prompt> is a character string displayed before the input area.

Usage:

INPUT takes entry from the keyboard as an expression of any data type. It then evaluates it placing the result into a newly created memory variable of the same data type. Pressing Return confirms entry. If Return is the only key pressed, INPUT terminates but does not create the memory variable. Note that Esc does not terminate INPUT.

Entering an invalid expression generates a runtime error.

Example:

```
INPUT "Expression: " TO exp
IF TYPE("exp") <> "U"
   ? exp
ELSE
   ? "No expression entered."
ENDIF
```

Library:

CLIPPER.LIB

See also:

ACCEPT, WAIT

JOIN

Syntax:

JOIN WITH <alias>/(<expC1>) TO <file>/(<expC2>) FOR <condition> [FIELDS <field list>]

Purpose:

To create a new database file by merging selected records and fields from two work areas.

Arguments:

<alias> is the work area to merge with records from the current work area.

<file> is the name of the target database file.

Options:

For: The FOR <condition> selects only records meeting the specified condition.

Fields: The <field list> is the projection of fields from both work areas into the new database file. To specify any fields in the secondary work area, reference them with the alias. If the FIELDS clause is not specified, all fields from the primary work area will be included in the target database file.

Usage:

JOIN is a projection of fields and a selection of records from two work areas that forms a new database file based on a general condition. JOIN works by making a complete pass through the secondary work area for each record in the primary work area evaluating the condition for each record processed in the secondary work area. When the condition is true (.T.), a new record is created in the target database file.

Note that the number of records processed will be the RECCOUNT() of the primary work area times the RECCOUNT() of the secondary work area. For example, if you have two database files with 100 records each, the number of records JOIN processes is the equivalent of sequentially processing a single database file of 10,000 records. All of which necessitates the maxim, "use this command with prudence."

Example:

The following example JOINs the Customer database file to the Invoices database file to produce a database file of Purchases:

```
USE Customers
SELECT 2
USE Invoices
SELECT 1
*
JOIN WITH Invoices TO Purchases;
   FOR Last = Invoices->Last;
   FIELDS First, Last, Invoices->Number,
Invoices->Amount
```

Library: CLIPPER.LIB

See also: APPEND FROM, REPLACE, SET RELATION

KEYBOARD

Syntax:

KEYBOARD <expC>

Purpose:

To stuff the keyboard buffer with a string.

Argument:

<expC> is the string to stuff into the keyboard buffer.

Usage:

KEYBOARD is used in combination with commands and functions that expect keyboard input as data and control information. These can be wait states such as ACCEPT, INPUT, and READ, or interface functions such as ACHOICE() and DBEDIT().

As a typical example, you can use KEYBOARD from within a SET KEY procedure to reassign keys in a wait state. Another use is within the ACHOICE() user function. Here you KEYBOARD the keys you want ACHOICE() to execute before returning control to it. The same concept applies to the DBEDIT() user function.

Note that each execution of KEYBOARD clears the keyboard buffer.

Example:

This example stuffs the keyboard with the escape key for three levels returning control to a main menu from three levels deep with one entry key. "Q" and a carriage return exits each menu, simulating a RETURN TO MASTER.

```
KEYBOARD "Q" + CHR(13) + "Q" + CHR(13) +
    "Q" + CHR(13)
```

Library:

CLIPPER.LIB

See also:

SET KEY, CHR(), LASTKEY(), NEXTKEY()

LABEL FORM

Syntax:

LABEL FORM <file1>/(<expC1>) [<scope>] [FOR <condition>] [WHILE <condition>] [SAMPLE] [TO PRINT] [TO FILE <file2>/(<expC2>)]

Purpose:

To display labels from a definition held in a (.lbl) file.

Argument:

<file1> is the name of the (.lbl) file that contains the FORM definition of the LABEL. If the extension is not specified (.lbl) is assumed.

Options:

Scope: The <scope> is the portion of the current database file to display labels. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to LABEL FORM within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Print: The TO PRINT clause echoes output to the printer.

File: The TO FILE clause echoes output to the specified filename, <file2>. If an extension is not specified, (.txt) is added.

Sample: The SAMPLE clause displays test labels as rows of asterisks. Each test label has the same number of columns and rows as the label definition. Then following each test label display is the query, "Do you want more samples?" Answering "Y" repeats the display of the test label. Answering "N" causes LABEL FORM to display the actual labels for the specified scope and condition.

Usage:

LABEL FORM displays labels using a definition stored in (.lbl) file. The (.lbl) can be created using RL.EXE. (See Chapter 12, *Clipper Utilities.*)

The contents of a LABEL field must be a valid expression. Note that Clipper does not support an expression list in the LABEL FORM contents. Anything following a comma in a LABEL FORM is ignored.

Example:

```
USE Sales INDEX Sales
LABEL FORM Sales TO PRINT
```

Library: CLIPPER.LIB

See also: REPORT FORM

LIST

Syntax:

LIST [OFF] [<scope>] <exp list> [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file>/(<expC>)]

Purpose:

To LIST the result of one or more expressions for each record processed.

Argument:

<exp list> is the list of values to display for each record processed.

Options:

Scope: The <scope> is the portion of the current database file to LIST. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to LIST within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Off: The OFF clause suppresses the display of record numbers.

Print: The TO PRINT clause echoes output to the printer.

File: The TO FILE clause echoes output to the specified filename. If an extension is not specified, (.txt) is added.

Usage:

LIST sends the results of the <exp list> to the screen in a tabular format with each column separated by a space. LIST is identical to DISPLAY with the exception that its default scope is ALL rather than NEXT 1.

Examples:

```
USE Sales
LIST DATE(), TIME(), Branch
LIST Branch, Salesman FOR Amount > 500;
   TO PRINT
```

One of the unique uses for LIST in Clipper is to provide the record movement, scope, condition, and end-of-file boundary tests for a procedure that traverses a database file. The procedure itself only processes the current record. In the example below, Report() operates only on the current record:

```
USE Sales
SET DEVICE TO PRINT
SET CONSOLE OFF
LIST Report() FOR Branch = "100"
SET CONSOLE ON
SET DEVICE TO SCREEN

FUNCTION Report
@ PROW() + 1, 3 SAY Branch
@ PROW() + 1, COL() + 1 SAY Branch
@ PROW() + 1, COL() + 1 SAY Amount
RETURN ""
```

To interrupt a LIST, use INKEY() as a part of the condition like the following example which terminates when you press Esc:

```
USE Sales
LIST Branch, Salesman, Amount FOR INKEY() <> 27
```

Library: CLIPPER.LIB

See also: DISPLAY

LOCATE

Syntax:

LOCATE [<scope>] FOR <condition> [WHILE <condition>]

Purpose:

To search for the first record in the current work area that matches the specified condition.

Argument:

<condition> specifies the next record to LOCATE within the given scope.

Options:

Scope: The <scope> is the portion of the current database file in which to perform the LOCATE. The default scope is ALL.

Condition: The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails where the LOCATE can scan. Note that after the first matching record is found, the WHILE condition is no longer operational.

Usage:

When you first execute a LOCATE, it searches from the beginning record of the scope for the first matching record in the current work area. It terminates when a match is found or the end of the LOCATE scope is reached. If it is successful, the matching record becomes the current record and FOUND() returns true (.T.). If it is unsuccessful, FOUND() returns false (.F.) and the positioning of the record pointer depends on the controlling scope of the pending LOCATE.

Each work area can have its own pending LOCATE scope which remains active until you execute another LOCATE in the current work area or the application is terminated.

LOCATE works in combination with CONTINUE. Any time later, you can resume the search from the current record pointer position with CONTINUE.

Examples:

```
USE Sales
? LASTREC()                          && Result: 84
LOCATE FOR Branch = "200"
? FOUND(), EOF(), RECNO()            && Result: .T. .F. 5
LOCATE FOR Branch = "5000"
? FOUND(), EOF(), RECNO()            && Result: .F. .T. 85
```

Library:

CLIPPER.LIB

See also:

CONTINUE, FIND, SEEK, FOUND(), STRTRAN()

MENU TO

Syntax:

MENU TO <memvar>

Purpose:

To execute a light-bar menu for the currently defined set of PROMPTs.

Argument:

<memvar> is where MENU TO places the result of the selection process. If the memory variable does not exist, MENU TO creates it as numeric type. If it does exist, MENU TO uses it to determine the currently highlighted choice in the list of pending PROMPTs.

Usage:

MENU TO is the selection mechanism for the Clipper light-bar menu system. Before executing it, paint the display of menu pads and define the associated MESSAGEs with a series @...PROMPT statements. Then invoke it with MENU TO <memvar>. If the <memvar> does not exist MENU TO creates it and places the highlight on the first PROMPT. If it does exist, its value determines the first PROMPT highlighted.

Navigation and selection: Pressing the arrow keys moves the highlight to the next or previous PROMPTs. As each PROMPT is highlighted the associated MESSAGE displays on the row specified with SET MESSAGE. If WRAP is ON, an Uparrow from the first PROMPT moves the highlight to the last PROMPT. Likewise, a Dnarrow from the last PROMPT moves the highlight to the first prompt. To make a selection, press Return or the first character of a PROMPT. MENU TO then returns the position of the selected PROMPT as a numeric value into the specified memory variable. Pressing Esc terminates the menu without making a choice and returns zero. The following table summarizes the active keys within MENU TO:

Table: 5-6 MENU TO Active Keys

Key	Action
Uparrow	Previous PROMPT
Dnarrow	Next PROMPT
Home	First PROMPT
End	Last PROMPT
Leftarrow	Previous PROMPT
Rightarrow	Next PROMPT
PgUp	Select PROMPT, returning position
PgDn	Select PROMPT, returning position
Return	Select PROMPT, returning position
Esc	Abort selection, returning zero
First letter	Select first PROMPT with same first letter, returning position.

SET KEY procedures: MENU TOs can be nested within SET KEY procedures without clearing the pending PROMPTs (unlike GET/READ). However, if the same memory variable is used for nested menus, it retains the previous value unless it is declared PRIVATE in the SET KEY procedure. Therefore, it is recommended that a different <memvar> be used for each menu. When you are in a SET KEY procedure, you can access the name of the return memory variable using READVAR(). This is useful as status information or to stuff a new PROMPT position into the calling menu.

Note that a maximum of 32 PROMPTs per menu are allowed.

Example:

```
SET MESSAGE TO 23 CENTER
@ 6, 10 PROMPT "Add"  MESSAGE "New acct"
@ 7, 10 PROMPT "Edit" MESSAGE "Change Acct"
@ 9, 10 PROMPT "Quit" MESSAGE "Return to DOS"
MENU TO Choice
```

Library: CLIPPER.LIB

See also: @....PROMPT, SET MESSAGE, SET WRAP, ACHOICE()

NOTE/*/&&

Syntax:

NOTE/* [<text>]/[<command>] && [<text>]

Purpose:

To place a comment as a command line or after a command statement in a program.

Argument:

<text> is a string of characters placed after the comment indicator.

Usage:

NOTE or asterisk (*) must begin the command, while double ampersand (&&) can occur anywhere on the command line. All characters after the comment indicator are ignored until Clipper encounters an end-of-line (carriage return/line feed). Comments, therefore, cannot be continued with the semicolon onto a new line.

Example:

```
NOTE This is a comment
* This is a comment
SET TALK ON                    && This is a comment

Note(s) This is a comment
*
* This is a comment
*
Note end
```

Library:

CLIPPER.LIB

PACK

Syntax:

PACK

Purpose:

To physically remove from the current database file records marked for deletion.

Usage:

When you PACK, all records marked for deletion are removed from the current database file, all indexes in USE in the current work area are REINDEXed, and the physical space occupied by the deleted records is recovered. Note that PACK does not create a backup file or use any temporary files. All file operations are internal to the current database file.

Example:

```
USE Sales
? LASTREC()                          && Result: 84
DELETE RECORD 4
PACK
? LASTREC()                          && Result: 83
```

Library:

CLIPPER.LIB

See also:

DELETE, RECALL, REINDEX, SET DELETED, ZAP, DELETED()

PARAMETERS

Syntax:

PARAMETERS <memvar list>

Purpose:

To identify memory variables that receive passed values or references.

Argument:

<memvar list> is one or more receiving memory variables separated by commas. The number of receiving variables does not have to match the number of parameters passed.

Usage:

Parameters are defined as either formal or actual. Formal parameters are the receiving memory variables specified as arguments of the PARAMETERS statement. Actual parameters are the arguments of the calling DO...WITH or user-defined function.

There are two methods of passing parameters, by value or by reference. By value means that the actual parameter is evaluated and the result is placed in a memory location. When the subsequent PARAMETERS executes, the value is transferred to the receiving variable. Passing by reference, by contrast, means that a pointer to the location of the actual parameter is passed instead of the value. Subsequent changes to the formal parameter are actually changes to the actual parameter, hence the term passing by reference.

Note that in Clipper there is no argument checking and therefore no requirement that the number of actual parameters match the number of formal parameters. To determine the number of actual parameters passed use PCOUNT().

Passing parameters to procedures and user-defined functions: The following rules apply when you pass memory variables and arrays to procedures and user-defined functions:

1. Memory variables and array identifiers are passed by reference to procedures. Array elements, expressions, variables within parentheses, and fields are passed by value (fields must be bounded by parentheses).

2. Parameters are passed by value to functions by default; however, they can be passed by reference if the variable is preceded by the "at" sign (@). Arrays are always passed by reference; array elements can only be passed by value.

Passing parameters from the DOS command line: You can pass multiple character strings to your program from the DOS command line. The character strings must be separated by spaces. A parameter bounded by quotes is passed as one string. For example:

C>PROG "CLIPPER COMPILER" 5

The routine that receives parameters from the DOS command should test the number of passed parameters with PCOUNT() to assure that critical parameters are defined.

Examples:

The following example passes parameters to a procedure. "Memvar" and "array" are passed by reference while "memvar2" is passed by value:

```
DECLARE array[1]
STORE "old" TO memvar, memvar2, array[1]
DO Proc WITH memvar, (memvar2), array
*
? memvar, memvar2, array[1]   && Result: new old new
*
RETURN

PROCEDURE Proc
PARAMETERS new_var, new_var2, new_array
STORE "new" TO new_var, new_var2, new_array[1]
RETURN
```

This example demonstrates passing parameters to user-defined functions.

```
STORE 10 TO memvar, memvar2
Modvar(@memvar, memvar2)
*
? memvar, memvar2             && Result: 20 10
*
RETURN

FUNCTION Modvar
PARAMETER new_var, new_var2
```

```
STORE 20 TO new_var, new_var2
RETURN ""
```

The following example demonstrates passing parameters from the DOS command line to the following program CLIPTEST:

```
* Cliptest.prg
PARAMETERS char, num, date, logical
?? TYPE("char")
?? TYPE("num")
?? TYPE("date")
?? TYPE("logical")
RETURN
```

Then executing CLIPTEST from DOS with the following command line:

C>CLIPTEST string 12 CTOD(SPACE(8)) .T.

produces the following results:

```
C C C C
```

When you call the same block of code as a procedure like this:

```
DO Cliptest WITH "string", 12, CTOD(SPACE(8)), .T.
```

the results are different:

```
C N D L
```

Library: CLIPPER.LIB

See also: DO, PRIVATE, PROCEDURE, PUBLIC, SET PROCEDURE, PCOUNT()

PRIVATE

Syntax:

PRIVATE <memvar list>

Purpose:

To hide memory variables declared PUBLIC or in higher-level procedures from the current and lower-level procedures. PRIVATE can also declare a private array and at the same time hide any PUBLIC array or private array of the same name from a higher-level procedure.

Arguments:

<memvar list> is the list of memory variables to hide or arrays to declare. The list can be any combination of memory variables and arrays.

Usage:

When a memory variable is declared PRIVATE in a procedure, the existing copy is hidden and not accessible until the declaring procedure terminates. PRIVATE, unlike PUBLIC, does not create a logical variable. Instead, the new variable is undefined. Once created, the scope of the new copy is the current procedure and therefore automatically released when the procedure terminates.

Arrays: In addition to declaring memory variables private, you can declare private arrays using the PRIVATE statement. Declaring an array PRIVATE works exactly the same as making any other memory variable PRIVATE; PUBLIC arrays and private arrays created in higher-level procedures are hidden. The array declaration using PRIVATE works the same as DECLARE with the exception that you can mix array and memory variable declarations. For more information on arrays, see DECLARE.

> **Note:** The ALL, LIKE, and EXCEPT clauses are not supported in Clipper.

Examples:

The following declares two arrays and three other variables PRIVATE:

```
PRIVATE array1[10], array2[20], var1, var2, var3
```

Library:

CLIPPER.LIB

See also:

DECLARE, PARAMETERS, PUBLIC

PROCEDURE

Syntax:

PROCEDURE <procedure>...RETURN

Purpose:

To identify the beginning of a procedure.

Argument:

<procedure> is the name of the procedure. The name can be up to 10 characters in length and must begin with an alpha character.

Each name must be unique from all other procedures, functions, programs, and formats within the same application.

Usage:

A PROCEDURE is any executable block of code beginning with the statement PROCEDURE <procedure name> and can occur anywhere in a program file, but cannot be nested within other procedures.

To terminate execution and branch back to the calling procedure, execute a RETURN. Unlike FUNCTIONs, a RETURN statement is not required. When compiling, Clipper assigns all statements from the PROCEDURE statement to the procedure name until another PROCEDURE statement, FUNCTION statement or end-of-file mark (1A hex) is encountered.

Example:

The following example demonstrates a main procedure calling two subprocedures that follow it within the same program file:

```
* Main.prg
DO Proc1
DO Proc2
RETURN

PROCEDURE Proc1
? "Proc one"            && Result: Proc one
RETURN

PROCEDURE Proc2
? "Proc two"            && Result: Proc two
RETURN
```

Library:

CLIPPER.LIB

See also:

DO, SET PROCEDURE

PUBLIC

Syntax:

PUBLIC <memvar list> [,clipper]

Purpose:

To declare memory variables and/or arrays as available to all procedures within a program.

Argument:

<memvar list> is the list of memory variables and/or arrays to declare PUBLIC. If an array is specified, it is created according to the dimension specified.

Options:

Clipper: To include Clipper extensions in a program and still allow the program to run under dBASE III PLUS the special memory variable, "clipper," is initialized to true (.T.) when declared PUBLIC. Using "clipper" as the argument of an IF...ENDIF construct blocks Clipper-specific code so these lines of code are not executed when run under dBASE III PLUS.

> **Note:** Compilation errors occur when the syntax of the dBASE III PLUS commands is not supported. In this instance, comment out statements you do not want compiled.

Usage:

Declaring a memory variable PUBLIC when the variable does not exist creates a new logical memory variable that is false (.F.). Once assigned a value, its scope is global and the variable becomes available to all procedures within the program. Note that you cannot declare an existing private variable PUBLIC. If you do this, Clipper simply ignores the declaration and the memory variable retains its private status and scope. Public variables, however, can be temporarily hidden from other procedures by declaring them PRIVATE.

Arrays: In addition to declaring PUBLIC memory variables, you can declare PUBLIC arrays. Declaring an array PUBLIC creates an array with the specified number of elements whose scope is all procedures in the current program. Note, however, that PUBLIC array elements are undefined until assigned a value.

For more information on arrays, see DECLARE.

Examples:

This example declares a series of memory variables and arrays PUBLIC:

```
PUBLIC var1, var2, array1[10], array2[10]
```

The following example demonstrates the different scoping behaviors of private and PUBLIC memory variables:

```
? TYPE("var1")       && Results: U
? TYPE("var2")       && Results: U
DO Proc1
? TYPE("var1")       && Results: C
? TYPE("var2")       && Results: U
RETURN

PROCEDURE Proc1
PUBLIC var1
var1 = "string1"
var2 = "string2"
RETURN
```

The following is an example of code using the PUBLIC memory variable "clipper" to allow Clipper enhancements in a program that runs under dBASE III PLUS without error:

```
PUBLIC clipper
@ 25, 10 SAY "Press any key to continue"
IF clipper
   key = INKEY(0)
ELSE
   key = INKEY()
   DO WHILE key = 0
      key = INKEY()
   ENDDO
ENDIF
```

Library:

CLIPPER.LIB

See also:

DECLARE, PARAMETERS, PRIVATE

QUIT/CANCEL

Syntax:

QUIT/CANCEL

Purpose:

To terminate program processing, close all open files, and return control to the operating system.

Usage:

QUIT or CANCEL can be used from anywhere in a program system to terminate and return to the operating system. A RETURN executed at the highest level procedure performs the same action.

Example:

The following example demonstrates a method of QUITting using a dialog box.

```
IF YesNoBox(10, 10, "Quit to DOS",;
            "BG+/B,B/W", "W/N,N/W", 2)
   QUIT
ENDIF
RETURN

FUNCTION YesNoBox
PARAMETERS r1, c1, strng, c_box, c_restore, choice
PRIVATE temp_scr
r2 = r1 + 6
c2 = c1 + LEN(strng)
SAVE SCREEN TO temp_scr
SET COLOR TO &c_box
@ r1, c1 CLEAR TO r1 + 6, c1 + LEN(strng) + 14
@ r1, c1 TO r1 + 6, c1 + LEN(strng) + 14 DOUBLE
@ r1 + 1, c1 + 3  SAY strng
@ r1 + 3, c1 + 3 TO r1 + 5, c1 + 8
@ r1 + 4, c1 + 4  PROMPT " Ok "
@ r1 + 3, c1 + 10 to r1 + 5, c1 + 19
@ r1 + 4, c1 + 11 PROMPT " Cancel "
choice = IF(choice  1 .OR. choice  2, 1, choice)
MENU TO choice
SET COLOR TO &c_restore
RESTORE SCREEN FROM temp_scr
RETURN (choice = 1)
```

Library:

CLIPPER.LIB

See also:

RETURN

READ

Syntax:

READ [SAVE]

Purpose:

To enter full-screen editing mode using the current pending GETs.

Option:

Save: The SAVE option retains the current set of pending GETs, allowing you to edit the same GETs by issuing another READ. If it is not specified, the current GETs are cleared.

Usage:

READ executes full-screen editing using all GETs pending since the most recent CLEAR, CLEAR GETS, CLEAR ALL or READ. If there is a FORMAT SET, READ passes control to it before entering full-screen edit.

Within a READ, you can edit the contents of and navigate between GETs. Whenever you press a key that terminates a GET, control passes to the VALID clause if it has been specified. A return value of true (.T.) terminates the GET and processes the navigation key. A false (.F.) value does not process the navigation key.

The following tables list active keys within a READ:

Table 5-7 Full-Screen Navigation Keys

Key	Action
Leftarrow, Ctrl-S	Character left. Does not move cursor to previous GET.
Rightarrow, Ctrl-D	Character right. At end of GET, cursor moves to next GET.
Ctrl-Leftarrow, Ctrl-A	Word left.
Ctrl-Rightarrow, Ctrl-F	Word right.
Uparrow, Ctrl-E	Previous GET.
Dnarrow, Ctrl-X, Return, Ctrl-M	Next GET.
Home	Beginning of GET.
End	Last character of GET.
Ctrl-Home	Beginning of first GET.
Ctrl-End	Beginning of last GET.

Table 5-8 Full-Screen Editing Keys

Key	Action
Del, Ctrl-G	Delete character at cursor position.
Backspace, Ctrl-H	Destructive backspace.
Ctrl-T	Delete word right.
Ctrl-Y	Delete from cursor position to end of GET.
Ctrl-U	Restore current GET to original value.

Table 5-9 Full-Screen Mode Keys

Key	Action
Ins, Ctrl-V	Toggle insert mode.

Table 5-10 Full-Screen Escape Keys

Key	Action
Ctrl-W, Ctrl-C, PgUp, PgDn	Terminate READ saving current GET
Return, Ctrl-M	Terminate READ from last GET
Esc	Terminate READ without saving current GET

In addition, you can terminate a READ by executing a CLEAR, CLEAR GETS, or CLEAR ALL from within a SET KEY procedure or a user-defined function initiated by VALID.

Note that READs found within a format are ignored. In Clipper only single screen page formats are supported.

Example:

```
STORE SPACE(10) TO var1, var2
@ 10, 10 SAY "Variable one: GET var1
@ 11, 10 SAY "Variable two: GET var2
READ
```

Library: CLIPPER.LIB

See also: @...GET, CLEAR GETS, SET FORMAT, LASTKEY()

RECALL

Syntax:

RECALL [<scope>] [FOR <condition>] [WHILE <condition>]

Purpose:

To reinstate records in the current database file that have been marked for deletion.

Options:

Scope: The <scope> is the portion of the current database file to RECALL. The default scope is the current record. If a condition is specified, the default scope becomes ALL.

Condition: The FOR clause specifies the conditional set of records to RECALL within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Usage:

If DELETED is ON, RECALL only reinstates the current record or a specific record if you specify the RECORD scope.

Example:

```
USE Sales
DELETE RECORD 4
? DELETED()                    && Results: .T.
RECALL
? DELETED()                    && Results: .F.
```

Network:

In a network environment, RECALL requires the current record be locked with RLOCK() if you are RECALLing a single record; the current database file be locked with FLOCK() or USEd EXCLUSIVEly, if you are RECALLing several records.

Library:

CLIPPER.LIB

See also:

DELETE, PACK, SET DELETED, ZAP, DELETED()

REINDEX

Syntax: REINDEX

Purpose: To rebuild all open indexes in the current work area.

Usage: When REINDEXing, the major consideration is the UNIQUE SETting. If you have UNIQUE ON, be aware that all indexes open in the current work area will be indexed UNIQUE regardless of how they were originally created. Remember, in Clipper UNIQUE is a global SETting and not an attribute of the index.

Network: In a network environment, REINDEX requires EXCLUSIVE USE of the current database file.

Library: CLIPPER.LIB

See also: INDEX, PACK, SET INDEX, SET UNIQUE, USE

RELEASE

Syntax:

RELEASE <memvar list>/[ALL[LIKE/EXCEPT <skeleton>]]

Purpose:

To delete memory variables.

Arguments:

<memvar list> is a list of memory variables to delete.

<skeleton> is a wildcard mask specifying a group of memory variables to delete or exclude from deletion.

Usage:

RELEASE deletes memory variables in two basic ways. If the scope of the deletion is a list, then the specified memory variables are deleted regardless of the memory variable scope unless they are hidden. This means that PUBLIC and private memory variables declared in higher level procedures are RELEASEd. If, however, the ALL clause is specified, then only memory variables defined in the current procedure are RELEASEd.

Note that a hidden memory variable does not become accessible when the local copy is RELEASEd but only when the procedure in which it was declared PRIVATE terminates.

Example:

```
PUBLIC one
one = "1"
DO Proc2
? TYPE("one")        && Result: C
RETURN

PROCEDURE Proc2
PRIVATE one
one = "2"
DO Proc3
? TYPE("one")        && Result: U
RETURN

PROCEDURE Proc3
RELEASE one          && Releases "one" from Proc2
? TYPE("one")        && Result: U
RETURN
```

Library:

CLIPPER.LIB

See also:

CLEAR ALL, CLEAR MEMORY, QUIT

RENAME

Syntax: RENAME <file1>.<ext> TO <file2>.<ext>

Purpose: To rename a file to a new name.

Arguments: <file1> is the name of the file to rename. The filename must include an extension if the file has one.

<file2> is the new filename including extension.

Example:

```
? FILE("Sales.dbf")                    && Result: .T.
? FILE("Oldsales.dbf")                 && Result: .F.
RENAME Sales.dbf TO Oldsales.dbf
? FILE("Oldsales.dbf")                 && Result: .T.
```

Library: CLIPPER.LIB

See also: COPY FILE, ERASE, RUN, FILE()

> **Warning:** The file to be RENAMEd must be CLOSEd before executing this command. Note that when a database file is RENAMEd, the associated memo (.dbt) file must be RENAMEd also.

REPLACE

Syntax:

REPLACE [<scope>] [<alias>->]<field1> WITH <exp1> [,<field2> WITH <exp2>,...] [FOR <condition>] [WHILE <condition>]

Purpose:

To change the contents of fields to the results of specified expressions.

Arguments:

<field> is the name of the target field to change. This can be a field of any type including memo. There is one target field per WITH clause.

<exp> is the replacement expression.

Options:

Alias: Fields in other work areas can be REPLACEd by preceding the field name with the alias.

Scope: The <scope> is the portion of the current database file to REPLACE. The default is the current record. Specifying a condition changes the default to ALL.

Condition: The FOR clause specifies the conditional set of records to REPLACE within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Usage:

Be aware that when you REPLACE a key field, the index is updated and the relative position of the record pointer within the index is changed. This means that REPLACEing a key field with a scope or a condition will yield an erroneous result. If this is necessary, CLOSE INDEX, perform the REPLACE, SET INDEX TO the appropriate index files, and then REINDEX.

Memo fields: Since Clipper supports the manipulation of memo fields in the same way as character fields, you can REPLACE a memo field with a character string.

Example:

```
SELECT 1
USE Customer
APPEND BLANK
SELECT 2
USE Invoices
*
REPLACE Charges WITH Customer->Markup * Cost,;
   Custid  WITH Customer->Custid,;
   Customer->Last_tran WITH DATE()
```

Network:

In a network environment, REPLACE requires the current record be locked with RLOCK() if you are REPLACEing a single record; the current database file be locked with FLOCK() or USEd EXCLUSIVEly if you are REPLACEing several records. If a field is being REPLACEd in an unselected work area, that record must also be locked with FLOCK().

Library:

CLIPPER.LIB

See also:

APPEND, JOIN, UPDATE, STRTRAN()

REPORT FORM

Syntax:

REPORT FORM <file1>/(<expC1>) [<scope>] [FOR <condition>] [WHILE <condition>] [TO PRINT] [TO FILE <file2>/(<expC2>)] [SUMMARY] [PLAIN] [HEADING <expC3>] [NOEJECT]

Purpose:

To display a tabular and optionally grouped report with page and column headings from a definition held in a (.frm) file.

Argument:

<file1> is the name of the (.frm) file that contains the FORM definition of the REPORT. If the extension is not specified (.frm) is assumed.

Options:

Scope: The <scope> is the portion of the current database file to report. The default scope is ALL.

Condition: The FOR clause specifies the conditional set of records to report within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Print: The TO PRINT clause echoes output to the printer.

File: The TO FILE clause echoes output to a file, <file2>, without form feed characters (ASCII 12). If a file extension is not specified, a (.txt) is added.

Summary: The SUMMARY clause causes the REPORT FORM to display only group, subgroup, and grand total lines. Detail lines are suppressed.

Plain: The PLAIN clause suppresses the display of the date, page number, and pagination. In addition, the report title and column headings display only at the top of the report.

Heading: The HEADING clause places the result of <expC3> on the first line of each page. <expC3> is evaluated only once at the beginning of the report and before the record pointer is moved.

Noeject: The NOEJECT clause suppresses the initial page eject when the TO PRINT clause is used.

Usage:

The REPORT FORM executes a report using a definition stored in a (.frm) file. The (.frm) can be created using RL.EXE. (See Chapter 12, *Clipper Utilities*.)

If you want to include form feed characters when sending the REPORT FORM TO FILE, add the lines of code shown below to your program:

```
SET PRINTER TO <file>
REPORT FORM <file> TO PRINT
SET PRINTER TO
```

Example:

```
USE Sales INDEX Sales
REPORT FORM Sales TO PRINT FOR Branch = "100"
```

Library:

CLIPPER.LIB

See also:

LABEL FORM, LIST

RESTORE

Syntax:

RESTORE FROM <file>/(<expC>) [ADDITIVE]

Purpose:

To retrieve memory variables from a memory (.mem) file.

Argument:

<file> is the memory (.mem) file to load from disk.

Option:

Additive: When the ADDITIVE option is specified, memory variables loaded from the memory file are added to the existing pool of memory variables. Memory variables with same names are overwritten unless hidden.

Without this clause, all existing memory variables are released before the memory file is loaded.

Usage:

When you RESTORE memory variables, they are initialized as private with the current procedure as the scope unless they are declared PUBLIC prior to the RESTORE and the ADDITIVE clause is specified.

Example:

The following example demonstrates a typical application of SAVE and RESTORE. Here memory variables containing screens are SAVEd TO and RESTOREd FROM memory files.

```
* Create and use a pseudo-array of screens.
SAVE SCREEN TO scr_1
SAVE ALL LIKE  scr_* TO Screens
*
<statements...>
*
RESTORE FROM Screens ADDITIVE
ptr = "1"
RESTORE SCREEN FROM scr_&ptr
```

Library:

CLIPPER.LIB

See also:

RESTORE SCREEN, SAVE, SAVE SCREEN

RESTORE SCREEN

Syntax: RESTORE SCREEN [FROM <memvar>]

Purpose: To redisplay a previously saved screen.

Options: **From:** The FROM clause redisplays a screen from the specified <memvar>.

Usage: RESTORE SCREEN is used generally in conjunction with the SAVE SCREEN to avoid repainting the original screen that has been temporarily replaced. Note, however, that <memvar> is an ordinary character variable and as such can be created and manipulated in any way you wish.

Example: The following example is a small pop-up box:

```
ans = .F.
SAVE SCREEN
@ 10, 10 CLEAR TO 12, 45
@ 10, 10 TO 12, 45 DOUBLE
@ 11, 12 SAY "File exists, overwrite (y/n)? ";
   GET ans PICTURE "Y"
READ
RESTORE SCREEN
RETURN
```

Library: CLIPPER.LIB

See also: RESTORE, SAVE SCREEN

RETURN

Syntax:

RETURN [<exp>]

Purpose:

To terminate a procedure or program returning control to either the calling procedure or the operating system.

Argument:

<exp> evaluates to the return value if the current program structure is a user-defined function.

Usage:

When RETURN is used in the highest level program, control passes to the operating system. In a procedure, Clipper returns to the calling procedure and releases all PRIVATE memory variables.

There can be more than one RETURN in a procedure or user-defined function. A procedure or user-defined function need not, however, end with a RETURN. Since user-defined functions must return values, each must contain at least one RETURN with an argument.

Note that Clipper does not support RETURN TO MASTER or any other form of RETURN specifying the level of the call to return to.

Examples:

```
PROCEDURE <procedure name>
*
<statements>...
*
RETURN

FUNCTION <function name>
*
<statements>...
*
RETURN <return expression>
```

Library:

CLIPPER.LIB

See also:

CANCEL, PRIVATE, PUBLIC

RUN/!

Syntax: RUN/! <DOS command>/(<expC>)

Purpose: To execute a DOS command or program from within a compiled application.

Argument: <DOS command> is any executable program including resident DOS commands and COMMAND.COM.

Usage: When you RUN a DOS program, Clipper executes another copy of COMMAND.COM passing the DOS command to run at the same time. This has two implications. First, you must have enough memory for COMMAND.COM (27K for DOS 3.2) and the program you wish to execute. Second, COMMAND.COM must be available on the path specified by COMSPEC (the default is the root directory of the disk where you boot DOS). If COMMAND.COM is not located on this disk or the disk is changed, SET COMSPEC to the new location prior to running the Clipper application.

Memory resident programs: You should not RUN memory resident programs from within Clipper since you may lose memory when the application terminates.

DOS Access: One of the options you may want to give your users is direct access to DOS. You can do this with the command:

```
RUN COMMAND
```

If you use this technique, you may also want to change the DOS prompt in order to give instructions on how to return to the application program. To set this up, create a batch file to set the DOS access prompt, load the application program, and restore the DOS prompt after the application program terminates as follows:

```
echo off
prompt Dir: $p$_Type EXIT to return.$_$g
<your application program>
prompt $p$g
```

Then instruct the user to execute the batch file in place of the application .EXE file.

Example:

The following example demonstrates how you can use RUN in combination with MEMOREAD() and MEMOWRIT() to create a user-defined function that calls your editor with the current memo field:

```
* Main.prg
success = EditorMemo("NE +", "Notes")

FUNCTION EditorMemo
PARAMETERS editor, memofld
IF MEMOWRIT("Clipedit.tmp", &memofld)
   RUN (editor + " Clipedit.tmp")
   REPLACE &memofld WITH MEMOREAD("Clipedit.tmp")
   RETURN 0
ELSE
   RETURN -1
ENDIF
```

EditorMemo() returns -1 if there is an error and zero if successful.

Library:

CLIPPER.LIB

See also:

INDEX, PACK, SET INDEX, SET UNIQUE, USE

SAVE

Syntax:

SAVE TO <file>/(<expC>) [ALL[LIKE/EXCEPT <skeleton>]]

Purpose:

To save memory variables to a memory (.mem) file.

Arguments:

<file> is the name of the file where specified memory variables are SAVEd. If no extension is specified, Clipper creates the file with a (.mem) extension.

<skeleton> is the wildcard mask to specify a group of memory variables to SAVE.

Usage:

SAVE copies the specified memory variables to a memory file without any reference to scope (public or private). Note that hidden memory variables are not SAVEd even if they are PUBLIC.

Note also that you cannot SAVE arrays to memory files.

Examples:

```
one = "1"
SAVE TO Temp
one = "2"
RESTORE FROM Temp
? one                    && Result: 1
```

Library:

CLIPPER.LIB

See also:

RESTORE, RESTORE SCREEN, SAVE SCREEN

SAVE SCREEN

Syntax:

SAVE SCREEN [TO <memvar>]

Purpose:

To SAVE the current SCREEN to a buffer or optional memory variable.

Options:

To: The TO clause specifies a memory variable to SAVE the current SCREEN. The variable created is character type and the length of the screen is 4000 bytes.

Usage:

Multiple screens may be saved by assigning each screen to a memory variable. SAVE SCREEN is used in conjunction with RESTORE SCREEN to eliminate the need to repaint an original screen that has been temporarily replaced.

In addition to ordinary memory variables, you can also SAVE SCREENs to array elements making screen management easier. Note, however, you cannot SAVE arrays to (.mem) files in order to SAVE multiple screens to disk.

Example:

```
DECLARE screens[10]
SAVE SCREEN TO screens[1]
*
<statements>...
*
RESTORE SCREEN FROM screens[1]
```

Library:

CLIPPER.LIB

See also:

RESTORE SCREEN, SAVE

SEEK

Syntax:

SEEK <exp>

Purpose:

To search an index for the first key matching a specified expression.

Arguments:

<exp> is an expression to be matched with an index key.

Usage:

SEEK searches the controlling index starting with the first key and proceeds until a match is found or there is a key value greater than the search argument. If there is a match, the record pointer is positioned to the record number found in the index. If SOFTSEEK is OFF (the default) and SEEK does not find a match, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). If SOFTSEEK is ON, the record pointer is positioned to the record with the first key value greater than the search argument and FOUND() returns false (.F.). In this case, EOF() only returns true (.T.) if there are no keys in the index greater than the search argument.

Example:

```
USE Sales INDEX Branch
? INDEXKEY(0)                          && Result: BRANCH
? Branch                               && Result: 100
SEEK "100"
? FOUND(), EOF(),   RECNO()            && Result: .T. .F. 1
```

Library:

CLIPPER.LIB

See also:

FIND, INDEX, LOCATE, SET DELETED, SET EXACT, SET INDEX, SET SOFTSEEK, USE, EOF(), FOUND(), RECNO()

SELECT

Syntax:

SELECT <work area>/<alias>/(<expN>)

Purpose:

To change the current work area.

Arguments:

<work area> is a number between zero and 254 inclusive. Zero designates the first available work area beginning with work area one.

<alias> is the name of an existing work area if there is a database file open in that area. The first 10 work areas can be referred to with the letters A through J. Specifying a non-existent alias produces a runtime error.

(<expN>) is a numeric expression that evaluates to a number between zero and 254. The expression must be bounded by opposing parentheses to be evaluated.

Usage:

Clipper supports 254 work areas. SELECTing 0 selects the first unused area. Within each work area, you can open a database file and up to 15 index files. Each work area has a number of attributes which include:

Table 5-11 List of Work Area Attributes

Attribute	Function
Alias name	ALIAS()
Work area number	SELECT()
Database file	ALIAS()
Index file(s)	INDEXORD()/INDEXKEY()
Exclusivity	FLOCK()/NETERR()
Filter condition	
Deleted filter	
Locate condition	
Relation(s)	
Number of records	LASTREC()/RECCOUNT()
Number of fields	FCOUNT()
Record number	RECNO()
Beginning-of-file flag	BOF()
End-of-file flag	EOF()
Found flag	FOUND()

Example:

```
last_area = SELECT()
SELECT 0
USE Newfile
*
<statements>...
*
SELECT (last_area)
```

Library: CLIPPER.LIB

See also: USE, SET INDEX, ALIAS(), SELECT()

SET ALTERNATE

Syntax:

SET ALTERNATE TO [<file>/(<expC>)]

SET ALTERNATE on/OFF/(<expL>)

Purpose:

To direct output from commands other than @...SAY...GET to a text file.

Options:

To: Specifying a <file> as the argument of the TO clause opens a standard ASCII text file with a default extension of (.txt). The filename can optionally include an extension, drive letter, and/or path. If a file with the same name exists, subsequent output will overwrite the current contents of that file.

Toggle: SET ALTERNATE on/OFF toggles the echo of output to the current alternate file. Note that when you SET ALTERNATE OFF, the alternate file is not closed.

Usage:

Alternate files are not related to work areas and only one may be open at a time. To close an alternate file, use CLOSE ALTERNATE, CLOSE ALL, or SET ALTERNATE TO with no argument. Note that @...SAYs cannot be echoed to a disk file using SET ALTERNATE, but SET PRINTER TO <filename> does support this feature.

Example:

```
SET ALTERNATE TO Listfile
SET ALTERNATE ON
USE Customers
DO WHILE .NOT. EOF()
   ? Lastname, City
   SKIP
ENDDO
USE
CLOSE ALTERNATE
```

Library:

CLIPPER.LIB

See also:

CLOSE, DISPLAY [TO FILE], LABEL FORM [TO FILE], LIST [TO FILE], REPORT FORM [TO FILE], SET PRINTER TO, TEXT [TO FILE], TYPE [TO FILE], FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

SET BELL

Syntax:

SET BELL on/OFF/(<expL>)

Purpose:

To toggle sounding of the bell during full-screen operations.

Usage:

The bell sounds in the following instances:

- You enter a character at the last character position in a GET.
- You attempt to enter an invalid data type into a GET. The data type is controlled by the data type of the field, memory variable, or by the PICTURE template.

Note that you can sound the bell explicitly using the command: ?? CHR(7).

Example:

```
SET BELL OFF
SET FORMAT TO No_bell
DO WHILE LASTKEY() <> 27
   READ
ENDDO
CLOSE FORMAT
SET BELL ON
```

Library:

CLIPPER.LIB

See also:

SET CONFIRM, CHR()

SET CENTURY

Syntax:

SET CENTURY on/OFF/(<expL>)

Purpose:

To toggle the year's century digits for the input and display of date fields and memory variables.

Usage:

When CENTURY is OFF, the century digits of dates are not displayed and cannot be input. If a calculation is performed on a date resulting in a non-twentieth century value, the date contains the correct century although it does not display. Dates entered without century digits default to the twentieth century.

When CENTURY is ON, a date field displays with a four-digit year and non-twentieth century dates can both be displayed and input.

Clipper supports all dates in the range 01/01/0100 to 12/31/2999.

Examples:

```
SET CENTURY OFF
? DATE()                    && Result: 09/01/87
SET CENTURY ON
? DATE()                    && Result: 09/01/1987
*
SET CENTURY (DATE() >= CTOD("01/01/2000")
```

Library:

CLIPPER.LIB

See also:

SET DATE, CTOD(), DATE(), DTOC(), DTOS(), YEAR()

SET COLOR

Syntax:

SET COLOR TO [<standard> [,<enhanced>] [,<border>] [,<background>] [,<unselected>]]/(<expC>)

Purpose:

To define colors for the next screen painting activity.

Options:

Standard/Enhanced: The "standard" and "enhanced" displays are color pairs with a foreground and an optional background color. "Standard" is used by all output, such as @...SAY and ?. "Enhanced" setting affects only the display of GETs.

Border: Sets border color.

Background: The "background" is not currently supported by any machines for which Nantucket provides drivers.

Unselected: The "unselected" option allows the current GET to be displayed in the "enhanced" setting while other GETs are the "unselected" color.

Attributes: High intensity and blinking are the attributes of colors. High intensity is denoted by "+" and blinking with "*". Each attribute specified is applied to the foreground color no matter where it occurs in the setting definition.

SET COLOR TO with no argument restores the default values which are: W/N,N/W,,,N/W.

Usage:

There are a number of colors supported for enhancing screen display. Each color is denoted by a letter or number. When you specify a color setting, numbers and letters should not be mixed. When numbers are used, the number to the left of the slash is written to the high order 4 bits of the color attribute byte, and the number to the right is written to the low order 4 bits.

The following table lists all the colors available:

Table 5-12 Clipper Color Table

Color	Number	Letter
Black	0	N
Blue	1	B
Green	2	G
Cyan	3	BG
Red	4	R
Magenta	5	RB
Brown	6	GR
White	7	W
Gray		N+
Yellow		GR+
Blank		X
Underline		U
Inverse Video		I

On monochrome monitors, color is not supported. Clipper, however, supports the monochrome attributes reverse video (I) and underlining (U).

Note: SET COLOR TO using numbers is not supported if you link ANSI.OBJ.

Color variables: To make managing color easier, consider the assignment of colors to memory variables. This will allow you to centralize the assignment of colors and implementation of runtime color configuration for users.

To set each color configuration, assign the complete SET COLOR argument as a character string to a single memory variable. Then use macro substitution or bound the color variable when you SET COLOR. For example:

```
c_says = "W/N,BG+/B,,,W/N"
SET COLOR TO &c_says
*
SET COLOR TO (c_says)
```

Note that in this case a list containing commas is allowed within a macro variable.

Since you may have a great number of color variables in a large system, you may also want to name color variables as a class by specifying each one with a leading identifier such as "c_." When you run your cross-reference utility, you will appreciate the grouping.

Examples:

The following demonstrates using a color setting to suppress the screen to enter a protected password:

```
SET COLOR TO W+/N, X
password = SPACE(6)
@ 13, 12 SAY  "Password: ";
   GET   password;
   VALID Password(password, "secret")
READ
SET COLOR TO
RETURN

FUNCTION Password
PARAMETERS getvar, word
IF getvar  word
   @ 22, 00
   WAIT "Bad password, press any key..."
   @ 23, 00
   RETURN .F.
ENDIF
RETURN .T.
```

The following example makes the current GET red on white while the rest are black on white.

```
color = "W/N,R/W,,,N/W"
SET COLOR TO &color
STORE SPACE(10) TO one, two, three
@ 1, 1 SAY "Enter One: " GET one
@ 2, 1 SAY "Enter Two: " GET two
@ 3, 1 SAY "Enter Three: " GET three
READ
```

Library: CLIPPER.LIB

See also: ISCOLOR()

SET CONFIRM

Syntax:

SET CONFIRM on/OFF/(<expL>)

Purpose:

To toggle between terminating the current GET when it is full or requiring a terminating key press.

Usage:

The following keys terminate GETs in Clipper:

Ctrl-Home
Ctrl-End
Uparrow
Dnarrow
Ctrl-C, PgUp
Ctrl-W, PgDn
Esc
Return

Note that CONFIRM has no effect on MENU TO.

Library:

CLIPPER.LIB

See also:

@...GET, READ, SET BELL

SET CONSOLE

Syntax: SET CONSOLE ON/off/(<expL>)

Purpose: To toggle the display of commands to the screen.

Usage: SET CONSOLE affects all output and prompts sent to the screen by all display commands including prompts for ACCEPT, INPUT, and WAIT. Full-screen commands such as @...SAY...GET, @...PROMPT, @...BOX, @...CLEAR TO, @...TO, and CLEAR display to the screen independent of the current CONSOLE SETting.

Example:

```
USE Sales
SET CONSOLE OFF
LIST Branch TO PRINT
SET CONSOLE ON
```

Library: CLIPPER.LIB

See also: SET DEVICE

SET CURSOR

Syntax:

SET CURSOR ON/off/(<expL>)

Purpose:

To toggle the screen cursor on and off.

Usage:

When the CURSOR is OFF, keyboard entry and screen display are unaffected. The cursor is merely hidden and data entry may still be accomplished without the cursor being visible.

The primary usefulness of this command is to suppress the cursor while the screen is being painted. Ideally, the only time the cursor should show in a production program is when you are in full-screen edit, the memo editor, or some kind of line edit.

Example:

```
SET CURSOR OFF
ans = .N.
@ 24, 0
@ 24, 15 SAY "Do you want to QUIT [Y/N]?";
   GET ans;
   PICTURE "Y"
READ
SET CURSOR ON
```

Library:

CLIPPER.LIB

See also:

SET CONSOLE

SET DATE

Syntax:

SET DATE
AMERICAN/ANSI/BRITISH/FRENCH/GERMAN/ITALIAN

Purpose:

To set the format of the date type for display, date function arguments, and date function return values.

Options:

The default SETting is AMERICAN.

Table 5-13 SET DATE Formats

SETting	Format
AMERICAN	mm/dd/yy
ANSI	yy.mm.dd
BRITISH	dd/mm/yy
FRENCH	dd/mm/yy
GERMAN	dd.mm.yy
ITALIAN	dd-mm-yy

Usage:

SET DATE is a global SETting and affects the behavior of dates throughout an application program. This allows you to control date formatting in a way that facilitates porting applications to foreign countries.

To configure a compiled application to the proper date setting at runtime, pass a DOS environmental variable to your application program, retrieve its value with GETE() and SET DATE with this value. For example, in DOS:

```
SET CLIP_DATE=BRITISH
```

Later in the configuration section of your application program:

```
date_set = UPPER(GETE("CLIP_DATE"))
DO CASE
CASE date_set = "AMERICAN"
   SET DATE AMERICAN
CASE date_set = "ANSI"
   SET DATE ANSI
CASE date_set = "BRITISH"
```

```
   SET DATE BRITISH
CASE date_set = "FRENCH"
   SET DATE FRENCH
CASE date_set = "GERMAN"
   SET DATE GERMAN
CASE date_set = "ITALIAN"
   SET DATE ITALIAN
ENDCASE
```

Examples:

```
SET DATE ANSI
? DATE()              && Result: 87.09.01
SET DATE BRITISH
? DATE()              && Result: 01/09/87
SET DATE FRENCH
? DATE()              && Result: 01/09/87
SET DATE GERMAN
? DATE()              && Result: 01.09.87
SET DATE ITALIAN
? DATE()              && Result: 01-09-87
SET DATE AMERICAN
? DATE()              && Result: 09/01/87
```

Library: CLIPPER.LIB

See also: SET CENTURY, CTOD(), DATE(), DTOC()

SET DECIMALS

Syntax:

SET DECIMALS TO <expN>

Purpose:

To set the number of decimal places displayed for the results of numeric functions and calculations.

Argument:

<expN> is the number of decimal places to display.

Usage:

The functions and arithmetic operations affected by SET DECIMALS are SQRT(), EXP(), LOG(), and division if SET FIXED is OFF. All other calculations display the number of decimal places that would normally result from the calculation. If SET FIXED is ON, all numerics obey the DECIMALS SETting.

Examples:

```
SET DECIMALS TO 2        && The default setting.
? 2/4                    && Result: 0.50
? 1/3                    && Result: 0.33
SET DECIMALS TO 4
? 2/4                    && Result: 0.5000
? 1/3                    && Result: 0.3333
```

Library:

CLIPPER.LIB

See also:

SET FIXED

SET DEFAULT

Syntax:

SET DEFAULT TO <drive> [:<path>]

Purpose:

To set the drive and directory where your Clipper application creates and save files.

Arguments:

<drive> identifies a disk drive name or letter.

<path> identifies the directory that you want to make the default. If you specify both a drive and directory, a colon must be included after the drive letter.

SET DEFAULT TO without an argument defaults to the last directory used on the designated drive.

Usage:

The default drive and directory are the drive and directory where your Clipper application is initially started. Once in the application, you can change this with SET DEFAULT.

Note that the default drive and directory do not constitute the search path for accessing files. To set a search path for file access, use SET PATH. SET DEFAULT is used primarily to specify the location where new files are written.

Note also that SET DEFAULT does not change the DOS drive and directory. Executing a RUN accesses the last drive and directory set in DOS.

Examples:

```
SET PATH TO
? FILE("Sales.dbf")                && Result: .F.
SET DEFAULT TO C:\CLIPPER\FILES
? FILE("Sales.dbf")                && Result: .T.
```

Library:

CLIPPER.LIB

See also:

SET PATH

SET DELETED

Syntax: SET DELETED on/OFF/(<expL>)

Purpose: To toggle automatic filtering of records marked for deletion.

Usage: When SET DELETED is ON, most commands ignore deleted records. If, however, you refer to a record by record number (GOTO or any command that supports the RECORD scope) the record displays even if marked for deletion. Additionally, SET DELETED ON has no affect on INDEX and REINDEXing.

RECALL ALL, however, honors SET DELETED and does not recall any records.

Example:

```
USE Sales
? LASTREC()        && Result: 84
DELETE RECORD 4
COUNT TO num
? num              && Result: 84
SET DELETED ON
COUNT TO num
? num              && Result: 83
```

Library: CLIPPER.LIB

See also: DELETE, INDEX, RECALL, SET INDEX, USE, DELETED()

SET DELIMITERS

Syntax:

SET DELIMITERS on/OFF/(<expL>)

SET DELIMITERS TO [<expC>]/DEFAULT

Purpose:

To define the character(s) used to delimit GETs.

Options:

Toggle: To display delimiters, SET DELIMITERS ON.

To: The TO <expC> clause defines a one or two character delimiter. Specifying one character places the same delimiter character as both the beginning and ending delimiters. Specifying two characters places the first character as the beginning delimiter and the second as the ending delimiter. Specifying DEFAULT or no delimiters SETs the delimiters back to colons.

Usage:

A space may be used in the character expression to suppress either the right or left (or both) delimiters.

Examples:

This example SETs DELIMITERS TO the character pair "[]":

```
mvar = SPACE(5)
SET DELIMITERS TO "[]"
SET DELIMITERS ON
@ 1, 0 SAY "Enter" GET mvar
READ
```

Result:

```
Enter [     ]
```

This second example SETs the DELIMITER preceding the GET to a colon and the DELIMITER that follows the GET to a space:

```
mvar = SPACE(5)
SET DELIMITERS TO ": "
SET DELIMITERS ON
@ 1, 0 SAY "Enter" GET mvar
READ
```

Result:

```
Enter :
```

Library: CLIPPER.LIB

See also: @...GET

SET DEVICE

Syntax:

SET DEVICE TO <SCREEN/print>

Purpose:

To direct the output of @...SAY commands to either the screen or the printer.

Options:

Screen: Specifying SCREEN as the DEVICE directs all @...SAYs to the screen and is independent of the SET PRINT and SET CONSOLE status.

Print: Specifying PRINT as the DEVICE directs all output to the device set with SET PRINTER TO. This can include a local printer port, a network spooler, or a file.

Usage:

When DEVICE is SET TO PRINT, @...SAY commands are sent to the printer and not echoed to the screen. @...GETs are ignored.

When sending @...SAYs to the printer, Clipper performs an automatic form feed whenever the current print head position is less than the last print position. An EJECT, however, resets the internal printer row and column values to zero. Additionally, you can specify new internal printer row and column values with SETPRC().

To send @...SAYs to a file, use SET PRINTER TO <filename> after SETting DEVICE TO PRINT.

Example:

```
SET DEVICE TO PRINT
@ 2,10 SAY "Hello there"
EJECT
```

Library:

CLIPPER.LIB

See also:

@...SAY, EJECT, SET PRINTER TO, PROW(), PCOL(), SETPRC()

SET ESCAPE

Syntax:

SET ESCAPE ON/off/(<expL>)

Purpose:

To toggle the ability to terminate a READ with Esc.

Usage:

If SET ESCAPE is ON, pressing Esc terminates a READ bypassing execution of VALID for the current GET.

SET ESCAPE OFF disables Esc from terminating a READ. SET KEY, however, still traps the key code for Esc so you can have a procedure that processes the key although it performs no action within a READ.

Note that SET ESCAPE OFF no longer disables program termination with Alt-C. This function is now performed by SETCANCEL().

Library:

CLIPPER.LIB

See also:

READ, SET KEY, SETCANCEL()

SET EXACT

Syntax:

SET EXACT on/OFF/(<expL>)

Purpose:

To determine how two character strings are compared.

Usage:

When EXACT is OFF two character strings are compared according to the following rules:

1. The strings are first considered equivalent, true (.T.).

2. Strings are then compared by character position for each character in both strings until the length of the string on the right side of the operator is exhausted or the comparison returns false (.F.).

This has two implications:

1. Two strings may equate even though they vary in length.

2. A string of length greater than zero can equate to a null string. This happens because a null string on the right side of the operator has a length of zero and so no comparisons are made. The result returned is then the initial value, true (.T.).

With SET EXACT ON, two strings must match exactly, except for trailing blanks. The same characters (upper and lower case) must be in the same order, and be of the same length to return true (.T.).

Examples:

```
SET EXACT OFF
? "123"   = "12345"        && Result: .F.
? "12345  = "123"          && Result: .T.
? "123"   = ""             && Result: .T.
? ""   = "123"             && Result: .F.

SET EXACT ON
? "123"   = "123  "        && Result: .T.
? " 123"  = "123"          && Result: .F.
```

Library:

CLIPPER.LIB

See also:

DISPLAY, FIND, LIST, LOCATE, SEEK, ==

SET EXCLUSIVE

Syntax:

SET EXCLUSIVE ON/off/(<expL>)

Purpose:

To toggle automatic exclusive access in a network environment for database, memo, and index files when a database file is brought into USE.

Usage:

When EXCLUSIVE is ON, all database and associated files are opened as non-sharable and cannot be USEd by other users until they are CLOSEd.

When EXCLUSIVE is OFF, all non-shared access must be controlled programmatically using RLOCK(), FLOCK(), and USE...EXCLUSIVE.

Examples:

See Chapter 10, *Using Clipper With A Local Area Network*, for examples.

Library:

CLIPPER.LIB

See also:

USE...EXCLUSIVE, FLOCK(), RLOCK()

SET FILTER

Syntax:

SET FILTER TO [<condition>]

Purpose:

To make a database file appear as if it contains only the records meeting a specified condition.

Argument:

<condition> is a logical expression that identifies a specific set of records.

To deactivate the filter condition, enter SET FILTER TO without a <condition>.

Usage:

When a FILTER condition is SET, the database file acts as if it contains only the records matching the specified condition. Each work area can have an active filter. Once a FILTER is SET, it is not activated until the record pointer is moved from its current position. Generally, use GO TOP to activate it.

As with SET DELETED, a filter has no effect on INDEX and REINDEX. You may, however, directly access filtered records with GOTO or commands using the RECORD scope.

Examples:

```
USE Customers
SET FILTER TO Zipcode > "50000"
GO TOP
LIST Lastname, Zipcode
SET FILTER TO
```

Library:

CLIPPER.LIB

See also:

SET DELETED

SET FIXED

Syntax:

SET FIXED on/OFF/(<expL>)

Purpose:

To toggle control of the display of numeric output by the current DECIMALs SETting.

Options:

On: When FIXED is ON, display of all numeric output is controlled by the DECIMALS SETting (two places if the SET DECIMALS default value is in effect).

Off: When FIXED is OFF, display of numeric output is dependent on the type of operation performed and according to the following rules:

Table 5-14 Numeric Decimal Display When FIXED Is OFF

Operation	Decimal Digits Displayed
Addition/subtraction	Same as operand with the greatest number of decimal digits
Multiplication	Sum of operand decimal digits
Division	SET DECIMALS
Exponentiation	SET DECIMALS
EXP(), LOG(), SQRT()	SET DECIMALS
VAL()	Same as operand

Examples:

```
SET DECIMALS TO 2
SET FIXED OFF
? 10.123456 + 10.12            && Result: 20.243456
? 10.123456 - 10.12            && Result: 0.003456
? 10.123456 * 10.12            && Result: 102.44937472
? 10.123456 / 10.12            && Result: 1.00
? 10.123456 **10.12            && Result: 14925450623.15
? EXP(10.123456)               && Result: 24920.75
? LOG(10.123456)               && Result: 2.31
? SQRT(10.123456)              && Result: 3.18
? VAL(STR(10.123456, 12, 7))   && Result: 10.1234560
```

Library:

CLIPPER.LIB

See also:

SET DECIMALS, EXP(), LOG(), SQRT(), VAL()

SET FORMAT

Syntax:

SET FORMAT TO <procedure>

Purpose:

To activate a format procedure so that whenever a READ is performed the format procedure is executed.

Argument:

<procedure> is a format (.fmt) file, a program (.prg) file, or a PROCEDURE.

SET FORMAT TO with no argument deactivates the current format.

Usage:

Format procedures are like any other procedure in Clipper differing only by the method you invoke them. Unlike the interpreted environment, formats are not opened at runtime and executed when the application is run. Instead, when Clipper encounters a SET FORMAT statement while compiling, it attempts to compile the specified file from disk as it would any other procedure reference. Because of this general character, format procedures can be stored in procedure files.

> **Note:** If you are compiling .CLP files with a program that contains format references, the accompanying format files must have (.prg) file extensions in order to compile correctly.

In Clipper there can only be one active format procedure unlike other dialects where each work area can have an active format.

Format declarations cannot be nested. Clipper ignores SET FORMAT statements within format procedures at compile time.

Clipper does not clear the screen before executing format files.

Commands, however, are allowed in format files, which gives more flexibility.

Clipper does not support multiple-page format procedures and so READ statements within format procedures are ignored.

Examples:

```
USE Sales
SET FORMAT TO Sales_scr
DO WHILE LASTKEY() <> 27
   APPEND BLANK
   READ
ENDDO
RETURN

PROCEDURE Sales_scr
CLEAR
@ 12, 12 SAY "Branch  : " GET  Branch
@ 13, 14 SAY "Salesman: " GET  Salesman
RETURN
```

Library: CLIPPER.LIB

See also: @...SAY...GET, READ

SET FUNCTION

Syntax:

SET FUNCTION <expN> TO <expC>

Purpose:

To assign to a function key a string of characters that are stuffed into the keyboard buffer when the key is pressed.

Argument:

<expN> is the function key number.

<expC> is the character string of up to 2000 characters submitted to the keyboard when the function key is pressed in a wait state.

Usage:

SET FUNCTION can assign to a function key a string to stuff the keyboard. This string may contain control characters, such as a Ctrl-C (which is the equivalent of a PgDn), to complete a READ.

> **Note:** A SET KEY definition takes precedence over a SET FUNCTION assignment for the same key.

Function keys between 2 and 40 inclusive can be assigned using this command. Function key 1, however, is reserved for use with Help.prg and cannot be assigned a string.

Table 5-15 List of Function Key Mappings

Function Key	Actual Key
1 - 10	F1 - F10
11 - 20	Shift-F1 - Shift-F10
21 - 30	Ctrl-F1 - Ctrl-F10
31 - 40	Alt-F1 - Alt-F10

Library:

CLIPPER.LIB

See also:

SET KEY

SET INDEX

Syntax: SET INDEX TO [<file list>/(<expC1>),...]

Purpose: To open the specified index file(s) in the current work area.

Argument: <file list> are the names of one or more (up to 15) index (.ntx or .ndx) files to open in the current work area, separated by commas.

Usage: SET INDEX assumes an (.ntx) or (.ndx) filename extension unless otherwise specified. You may include a drive letter and/or path name with the index filename.

CLOSE INDEX or SET INDEX TO without a filename closes all indexes open in the current work area.

When more than one index file is opened for the active database file, the first index becomes the controlling index. The record pointer is then positioned at the first logical record in the index. During database file processing, all open indexes are updated whenever a key value is appended or changed. To change the controlling index without issuing another SET INDEX command, use SET ORDER.

Macro variables: Index files may be specified by using macro variables. Each file listed, however, must be a separate variable. For example, this is not permissible:

```
ntx_list = "Name, Account"
SET INDEX TO &ntx_list
```

This formulation, however, is permissible:

```
ntx_1 = "Name"
ntx_2 = "Account"
SET INDEX TO &ntx_1, &ntx_2
```

Note that macros substituting a null string ("") or spaces are ignored.

Library: CLIPPER.LIB

See also: CLOSE, INDEX, REINDEX, SET ORDER, USE

SET INTENSITY

Syntax:

SET INTENSITY ON/off/(<expL>)

Purpose:

To toggle the display of GETs between enhanced and standard color settings.

Usage:

When INTENSITY is OFF, GETs appear in the same color as SAYs, the standard color setting. If INTENSITY is ON (the default), GETs appear in the enhanced color setting. Note that INTENSITY has no effect on other display commands and functions such as MENU...TO, ACHOICE(), and DBEDIT().

See also:

@...SAY...GET, SET COLOR

SET KEY

Syntax:

SET KEY <expN> TO [<procedure>]

Purpose:

To allow a procedure to be executed from any wait state when a designated key is pressed.

Arguments:

<expN> is the INKEY() value of the designated key. See Appendix G for a complete list of key values.

<procedure> is the procedure that executes when the assigned key is pressed. If the procedure is not specified the current key redefinition is released.

Usage:

A wait state is defined as any command that pauses program execution, such as:

WAIT
READ
ACCEPT
INPUT
MENU TO

Note that INKEY() is not a wait state.

A maximum of 32 keys may be SET at one time. At start-up, the system assumes:

```
SET KEY 28 TO Help      && F1 is help
```

Like Help.prg, three automatic parameters are passed to the SET KEY procedure. The parameters are "calling program," "line number," and "input variable." "Calling program" and "input variable" are both character type and "line number" is numeric type.

As a rule, CLEAR or READ should not be used in SET KEY procedures if the wait state is READ since both commands clear the pending GETs in the calling program. To clear the screen, use @ 0, 0 CLEAR instead.

Note: SET KEY takes precedence over SET FUNCTION for the definition of a function key.

Example:

```
SET KEY -1 TO Acct_hlp     && F2 shows account IDs.
USE Invoice
APPEND BLANK
@ 1,0 SAY "Press F2 for a list of" + ;
     "Account ID numbers"
@ 2,0 SAY "Enter Account ID ";
    GET ID
READ

* Lists account IDs when F2 is pressed.
PROCEDURE Acct_hlp
PARAMETERS p1, p2, p3      && Proc, line, and memvar.
SAVE SCREEN
@ 0, 0 CLEAR
SELECT 2
USE Accounts INDEX Acct_Id
DISPLAY ALL Company, Id
WAIT "Press any key to continue editing..."
USE
SELECT Invoice
RESTORE SCREEN
RETURN
```

Library: CLIPPER.LIB

See also: KEYBOARD, SET FUNCTION, LASTKEY()

SET MARGIN

Syntax:

SET MARGIN TO <expN>

Purpose:

To set the left margin for all printed output.

Argument:

<expN> defines the column position of the left margin on printed output.

Usage:

SET MARGIN has no effect on the screen display but does affect the beginning column position of all commands that output to the printer including REPORT and LABEL FORM.

If a SET MARGIN has not been executed, the left margin defaults to zero.

Example:

```
USE Sales
SET MARGIN TO 5
LIST Branch, Salesman TO PRINT
```

Library:

CLIPPER.LIB

See also:

@...SAY...GET, SET DEVICE, SET PRINT

SET MESSAGE

Syntax:

SET MESSAGE TO [<expN> [CENTER]]

Purpose:

To set the screen row where @...PROMPT...MESSAGEs display.

Argument:

<expN> specifies the row position where the messages display.

Messages appear on row <expN>, column 0 unless the CENTER option is used. The display of messages is suppressed by SET MESSAGE TO 0 or SET MESSAGE TO without an argument.

Options:

Center: The CENTER option centers the message on the specified row.

Example:

```
SET MESSAGE TO 23 CENTER
@ 5, 5 PROMPT "One" MESSAGE "Choice one"
@ 6, 5 PROMPT "Two" MESSAGE "Choice two"
MENU TO choice
```

Library:

CLIPPER.LIB

See also:

@...PROMPT, MENU, SET WRAP

SET ORDER

Syntax:

SET ORDER TO [<expN>]

Purpose:

To identify the specified open index as the controlling index.

Argument:

<expN> specifies the index to make the controlling index by pointing to its position in the list of open indexes in the current work area. This number can be in the range of zero to 15.

SETting ORDER TO 0 restores the database file to natural order (record number order) while leaving all indexes open.

Usage:

When you SET ORDER TO a new controlling index, all indexes are properly updated when you either append or edit records. Unlike dBASE III PLUS, this includes SET ORDER TO 0. After a change of controlling indexes, the record pointer still points to the same record number in the new index allowing you to switch orders efficiently.

To return the ordinal position of the current controlling index, use INDEXORD().

Examples:

```
USE Customers
INDEX ON Lastname TO Names
INDEX ON City + State TO Region
SET INDEX TO Names, Region
*
SET ORDER TO 2
? INDEXKEY(INDEXORD())      && Result: City + State
SET ORDER TO 0
? INDEXKEY(INDEXORD())      && Result: null value
SET ORDER TO 1
? INDEXKEY(INDEXORD())      && Result: Lastname
```

Library:

CLIPPER.LIB

See also:

INDEX, REINDEX, SET INDEX, USE, INDEXEXT(), INDEXKEY(), INDEXORD()

SET PATH

Syntax:

SET PATH TO [<path list>]

Purpose:

To specify the search path that Clipper follows when attempting to access files.

Arguments:

<path list> identifies the paths where Clipper searches for a file if it is not found in the current directory. A path is a pointer to a directory. It includes a list of all the directories from the root to the specified directory separated by backslashes. A path list then is the sequence of paths to search, each separated by a comma or semi-colon.

> Note: Because the semi-colon is a path list separator, continuation of a path command line with a semi-colon is not supported.

SET PATH TO with no argument releases the path list and Clipper only searches the current directory.

Usage:

SET PATH allows Clipper to find and update existing files in another drive and/or directory. When you attempt to access a file Clipper first looks for it in the current drive and directory. The current disk drive and directory is established when your Clipper application is loaded, or by SET DEFAULT. If the file is not found, Clipper then searches the path list you specified path by path until the first instance of the file is found. Note that FILE() respects the PATH SETting when searching for the existence of a file.

The path, however, is only for finding existing files. If you wish to create new files in another drive or directory, use SET DEFAULT TO <directory> or explicitly declare the path when specifying the filename.

To configure a compiled application to a site-specific path setting at runtime, pass a DOS environmental variable to your application program, retrieve its value with GETE(), and SET PATH with this value. For example, in DOS

C> SET CLIP_PATH=C:\APPS\DATA, C:\APPS\PROGS

Later in the configuration section of your application program:

```
path_set = GETE("CLIP_PATH")
SET PATH TO &path_set
```

Examples:

The following example is an acceptable PATH statement:

```
SET PATH TO A:\INVENTORY; B:\VENDORS
```

Because the semi-colon is part of the syntax of the path definition, the following example is not acceptable:

```
SET PATH TO A:\INVENTORY,;
   B:\VENDORS
```

Library:

CLIPPER.LIB

See also:

DIR, SET DEFAULT, FILE()

SET PRINT

Syntax:

SET PRINT on/OFF/(<expL>)

Purpose:

To toggle echo of output from commands other than @...SAY to the printer.

Usage:

When PRINT is ON, output sent to the screen is also sent to the printer unless CONSOLE is OFF. @...SAYs, however, are not affected by SET PRINT ON. To send them to the printer, use SET DEVICE TO PRINT.

Note that a number of commands such as REPORT and LABEL FORM can direct output to the printer from the respective command line. To suppress output to the screen, however, you must still SET CONSOLE OFF.

Example:

```
USE Customers
SET PRINT ON
SET CONSOLE OFF
DO WHILE .NOT. EOF()
   ? Customer
   SKIP
ENDDO
EJECT
SET PRINT OFF
SET CONSOLE ON
CLOSE DATABASES
RETURN
```

Library:

CLIPPER.LIB

See also:

EJECT, SET CONSOLE, SET DEVICE, SET PRINTER

SET PRINTER

Syntax:

SET PRINTER TO [<device>/<file>/(<expC>)]

Purpose:

To determine the destination of printed output.

Arguments:

<device> sends the printed output to the network or the local device. For some networks, the work station's printer should first be redirected to the file server (usually by running the network spooler program).

<file> sends all printer output including @...SAYs to the specified filename. If you do not specify a file extension, Clipper appends a (.prn) extension to the filename.

SET PRINTER TO with no arguments closes the print spool file and resets the default destination.

SETting PRINTER TO a non-existing device creates a file with the name of the device.

Usage:

The default device is PRN.

SET PRINTER directs printed output from any print related command to the specified device. Device names include LPT1, LPT2, LPT3 (all parallel ports), COM1, and COM2 (serial ports).

> **Note:** When specifying device names, be sure not to specify a trailing colon.

This capability has several uses:

- You can swap ports for managing multiple printers.
- You can direct output to a file for printing later or transfer to a remote computer via telecommunications.
- You can empty the printer spooler and reset the default device.

Examples:

This example directs printer output to LPT1 and empties the print spooler when it completes:

```
SET PRINTER TO LPT1
<Printing statements>...
SET PRINTER TO              && Empties print spooler.
```

The following example sends printer output to a text file overwriting the existing file if there is one:

```
SET DEVICE TO PRINT
SET PRINT ON
SET PRINTER TO Prnfile.txt
@ 0, 0 SAY "This goes to Prnfile.txt"
? "So will this!"
SET PRINTER TO              && Closes file.
SET DEVICE TO SCREEN
SET PRINT OFF
```

Library:

CLIPPER.LIB

See also:

@....SAY, SET DEVICE, SET PRINT

SET PROCEDURE

Syntax:

SET PROCEDURE TO [<file>]

Purpose:

To compile all procedures and user-defined functions within the specified file into the current object file.

Argument:

<filename> is the name of the procedure file. If the file has no extension, then (.prg) is assumed.

Usage:

A procedure file may contain any number of procedures or user-defined functions. When a SET PROCEDURE TO command is encountered, Clipper compiles the procedures and user-defined functions in the file. Essentially, a procedure file in Clipper operates much as an include directive does in other compiled environments.

CLOSE PROCEDURE and SET PROCEDURE TO with no argument are both ignored when encountered.

Example:

```
SET PROCEDURE TO Strlib
SET PROCEDURE TO Winlib
SET PROCEDURE TO Scrlib
SET PROCEDURE TO Appslib
DO Main
RETURN
```

Library:

CLIPPER.LIB

See also:

DO, FUNCTION, PROCEDURE, RETURN

SET RELATION

Syntax:

SET RELATION [ADDITIVE] TO [<key exp1> /RECNO()/<expN1> INTO <alias1>/<expC1>] [,TO <key exp2>/RECNO()/<expN2> INTO <alias2>/<expC2>]...

Purpose:

To relate two work areas using a key expression, record number, or numeric expression.

Arguments:

<key exp> is an expression used to perform a SEEK in the child work area each time the record pointer moves in the parent work area. For this to work, the child work area must have an index in USE.

RECNO() relates the parent to the child work area using the parent record number to perform a GOTO to the same record number in the child work area each time the record pointer moves in the parent work area. For this type of relation, it is recommended that the child work area not have an index in USE.

<expN> is an expression used to perform a GOTO to the matching record number in the child work area each time the record pointer moves in the parent work area. For this type of relation to execute correctly, the child work area must not have an index in USE.

<alias> identifies the child work area.

To release all RELATIONs in a work area, specify SET RELATION TO with no arguments.

Clipper supports eight relations per work area. Note that cyclical relations are not supported. You cannot relate a database file either directly, or indirectly, to itself.

Option:

Additive: The ADDITIVE clause adds the specified child relations to the relations already set in the current work area. If this clause is not specified, existing relations in the current work area are released before the newly specified child relations are set.

Usage:

A relation defined by SET RELATION causes the record pointer to move in the child work area in accordance with the movement of the record pointer in the parent work area. If there is not a match in the child work area, the child record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

> **Note:** SET RELATION does not obey SOFTSEEK and always behaves as if SOFTSEEK is OFF. If a match is not found in the child work area, the child record pointer is positioned to LASTREC() + 1.

Examples:

The following example relates three work areas in a multiple parent-child configuration with Customer related to both Invoices and Zip.

```
SELECT 0
area = SELECT()
SELECT (area + 1)
USE Invoices INDEX Invoices
SELECT (area + 2)
USE Zip INDEX Zipcode
SELECT (area)
USE Customer
SET RELATION TO Customer INTO Invoices,;
   TO Zipcode INTO Zip
*
LIST Customer, Street, Zip->City,;
   Invoices->Number, Invoices->Amount
```

Sometime later, you can add a new child relation using the ADDITIVE clause, like this:

```
SELECT 0
USE BackOrder INDEX BackOrder
SELECT Customer
SET RELATION ADDITIVE TO Cust_num INTO BackOrder
```

Library:

CLIPPER.LIB

See also:

INDEX, REPLACE, SET INDEX, SET ORDER, UPDATE, USE

SET SCOREBOARD

Syntax:

SET SCOREBOARD ON/off/(<expL>)

Purpose:

To toggle the display of messages within READ and MEMOEDIT() ON or off.

Usage:

When SCOREBOARD is ON, messages from modes display on line 0. These include the RANGE clause error message, the abort query message for MEMOEDIT(), and a message indicating insert mode both within MEMOEDIT() and READ.

Library:

CLIPPER.LIB

SET SOFTSEEK

Syntax:

SET SOFTSEEK on/OFF/(<expL>)

Purpose:

Toggles for "relative" SEEKing (looking for the next higher key value when a search fails using SEEK).

Usage:

If SOFTSEEK is ON and a match for the SEEK argument is not found, the record pointer is set to the next record in the index with a higher key value than the search key. If there is no record with higher key value, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

If SOFTSEEK is OFF and a SEEK is unsuccessful, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

Note that SET EXACT has no effect on a SEEK if SOFTSEEK is ON. Note also that SET RELATION ignores SOFTSEEK and so updates the record pointer in child work areas as if SOFTSEEK is OFF.

Example:

```
SET SOFTSEEK ON
SEEK "Findit"
IF FOUND()
    ? "found", RECNO()
ELSE
    ? "not found"
    ? EOF()
    ? RECNO()
ENDIF
```

Library:

CLIPPER.LIB

See also:

INDEX, SEEK, SET EXACT, SET RELATION

SET TYPEAHEAD

Syntax:

SET TYPEAHEAD TO <expN>

Purpose:

To set the size of the keyboard buffer.

Argument:

<expN> determines the number of characters the keyboard buffer can hold from a minimum of zero to a maximum of 32,768.

Note that setting the keyboard buffer size to zero may disable Alt-C and Alt-D if you are performing a very tight loop operation.

Library:

CLIPPER.LIB

See also:

ACCEPT, INPUT, KEYBOARD, READ, SET KEY, LASTKEY()

SET UNIQUE

Syntax:

SET UNIQUE on/OFF/(<expL>)

Purpose:

To toggle the inclusion of non-unique keys when creating a new index.

Usage:

When you INDEX with UNIQUE ON, Clipper creates an index with uniqueness as an attribute. As indexing proceeds and two or more records have the same key value, Clipper includes only the first record in the index. Whenever the unique index is updated, REINDEXed, or PACKed, only unique records are added. This happens without regard to the current UNIQUE SETting.

Note that this differs from previous versions of Clipper where UNIQUE was a global SETting and applied to the creation and updating of all open indexes.

Examples:

The following example lists all Customers by State:

```
SET UNIQUE OFF
USE Customers
INDEX ON State + Customer TO State
LIST State, Customer
```

This example lists all States where there are Customers:

```
SET UNIQUE ON
USE Customers INDEX ON State TO State
LIST State
```

Library:

CLIPPER.LIB

See also:

FIND, INDEX, REINDEX, SEEK, SET INDEX, USE

SET WRAP

Syntax: SET WRAP on/OFF/(<expL>)

Purpose: To toggle wrapping in MENUs.

Usage: When WRAP is ON and the highlight is on the last menu option, Rightarrow or Dnarrow moves it to the first option. If the first option is highlighted, Leftarrow or Uparrow moves it to the last option.

Library: CLIPPER.LIB

See also: @...PROMPT, MENU TO, SET MESSAGE

SKIP

Syntax:

SKIP <expN1> [ALIAS <work area>/<alias>/(<expN2>)]

Purpose:

To reposition the record pointer in the currently selected or specified work area relative to the current pointer position.

Argument:

<expN1> specifies the number of records to move the record pointer from the current position. A positive value moves the record pointer forward and a negative value moves the record pointer backwards. A zero argument flushes the current work area buffer to DOS buffers.

Option:

Alias: The ALIAS clause moves the record pointer in the designated work area instead of the current work area.

Usage:

SKIP without a numeric argument moves the current record position to the next record. SKIPping backward beyond the beginning-of-file moves the pointer to the first record and BOF() returns true (.T.). SKIPping forward beyond the end-of-file positions the record pointer at LASTREC() + 1 and EOF() returns true (.T.).

If an index file is in use, SKIP moves the specified number of positions as defined by the index.

Flushing Clipper buffers: Any command that causes a record to be read into Clipper's internal buffers flushes the current database file buffer to a DOS buffer if a change has been made to that buffer's contents since it was read from disk. SKIP 0 can be used to force a flush of the current database file buffer without causing record movement. Note that the specific time information is written to disk depends on DOS unless you execute a COMMIT to force an actual disk-write. SKIP 0 itself does not necessarily cause a disk write. Note also that the command does not flush INDEX buffers.

Examples:

```
USE Customers
? RECNO()                    && Result: 1
SKIP
? RECNO()                    && Result: 2
SKIP 10
? RECNO()                    && Result: 12
SKIP -5
? RECNO()                    && Result: 7
SKIP -10
? RECNO()                    && Result: 1
```

The following example moves the record pointer in a remote work area:

```
SKIP ALIAS 4
```

This is the same as:

```
SELECT 4
SKIP
SELECT 1
```

Library: CLIPPER.LIB

See also: COMMIT, CONTINUE, GOTO, FIND, LOCATE, SEEK, BOF(), EOF(), RECNO()

SORT

Syntax:

SORT [<scope>] ON <field1> [/A][/C][/D] [,<field2> [/A][/C][/D]]...
TO <file>/(<expC>) [FOR <condition>] [WHILE <condition>]

Purpose:

To copy records from the current work area to another database file in sorted order.

Arguments:

<field1...fieldN> are the fields to use as sort keys. Note that SORT does not work with substrings and expressions as keys.

<file> is the name of the target file for the sorted records. Unless otherwise specified, the new file is assigned the (.dbf) extension.

Options:

Order: SORT supports three sorting options:

- /A sorts in ascending order.
- /D sorts in descending order.
- /C sorts ignoring the case of the specified character field.

Unless otherwise specified, the default SORT is in ascending order.

Scope: The <scope> is the portion of the current database file to SORT. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to SORT within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Usage:

Clipper SORTs character fields in accordance with the ASCII value of each character within the string. Numeric fields are SORTed in numeric order, and date fields are SORTed chronologically. Logical and memo fields cannot be SORTed.

SORT performs as much of its operation as possible in memory and then it spools to a temporary disk file (Clipsort.tmp). This temporary file can be as large as the size of the source database file. Note also that a SORT uses up three file handles: the source database file, the target database file, and the temporary file.

Deleted source records: SORT copies deleted records to the target database file and the deleted records retain their deleted status.

Network: In a network environment, the database file to be SORTed should be locked with FLOCK() or USEed EXCLUSIVEly.

Library: CLIPPER.LIB

See also: INDEX, ASORT()

STORE

Syntax:

STORE <exp> TO <memvar list>/<memvar> = <exp>

Purpose:

To initialize and/or assign a value to one or more memory variables.

Arguments:

<exp> is a value of any data type assigned to the target memory variable(s).

<memvar list> are the memory variables to initialize and/or assign values. Memory variable names can be up to 10 characters in length and contain letters, numbers, and underscores (_). The first character, however, must be a letter.

Usage:

STORE both creates and assigns values to memory variables. Unlike other languages, there is no type declaration necessary. Clipper automatically assigns the data type based on the resulting value of the expression before the value is stored. The scope of a memory variable is private unless it is explicitly declared PUBLIC prior to initialization. A private memory variable is released when the procedure where it is initialized terminates with a RETURN. RELEASE, CLEAR MEMORY, and CLEAR ALL also release memory variables.

Fields and memory variables can have the same name. When there is a name conflict, field names take precedence unless the memory variable is identified with the memory variable alias M-><memvar> or you have compiled with the (-v) switch. The (-v) switch changes this so that memory variables have precedence over field names.

In Clipper, the maximum number of memory variables that can exist at one time is 2048. Arrays, however, count only as one memory variable, each containing up to 2048 elements (memory permitting).

Assigning a memo field to a non-existent memory variable creates a memory variable of character type.

Examples:

```
STORE "string" TO var1, var2, var3
var1 = "string2"
var2 = M->var1
```

Library:

CLIPPER.LIB

See also:

CLEAR MEMORY, PRIVATE, PUBLIC, RELEASE, RESTORE, SAVE

SUM

Syntax:

SUM [<scope>] <expN list> TO <memvar list> [FOR <condition>] [WHILE <condition>]

Purpose:

To sum a series of numeric expressions to memory variables for a range of records in the current work area.

Arguments:

<expN list> is the list of numeric values to SUM for each record processed.

<memvar list> identifies the receiving memory variables for the SUM and are created at the time of execution of the command. Existing memory variables with the same names are overwritten. This list must contain the same number of elements as the list of expressions to SUM.

Options:

Scope: The <scope> is the portion of the current database file to SUM. The default scope is ALL.

Condition: The FOR clause specifies the conditional set of records to SUM within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Examples:

```
USE Sales
SUM Amount + 10, Amount TO sum1, sum2
? sum1                && Result: 151515.00
? sum2                && Result: 150675.00
```

Library:

CLIPPER.LIB

See also:

AVERAGE, TOTAL

TEXT

Syntax:

TEXT [TO PRINT] [TO FILE <file>]
<text>...
ENDTEXT

Purpose:

To display a block of text to the screen optionally echoing output to the printer and/or a text file.

Arguments:

<text> is the block of literal characters to display to the screen. Text is displayed exactly as formatted.

Options:

Print: The TO PRINT clause echoes the display to the printer.

File: The TO FILE clause echoes the display to the specified <file>. If no extension is specified, (.txt) is added.

Usage:

Macro variables found within TEXT...ENDTEXT are expanded. Note, however, TEXT...ENDTEXT has no provision for word-wrapping. Text is displayed exactly as encountered.

Example:

```
name = "Ed Frisbee"
*
TEXT
Dear Mr. &name.:

It has come to our attention that your recent
business activities are of a dubious nature.
ENDTEXT
```

Results:

```
Dear Mr. Ed Frisbee:

It has come to our attention that your recent
business activities are of a dubious nature.
```

Library:

CLIPPER.LIB

See also:

?/??, @...SAY, MLCOUNT(), MEMOLINE()

TOTAL

Syntax:

TOTAL ON <key exp> [<scope>] [FIELDS <field list>] TO <file>/(<expC>) [FOR <condition>] [WHILE <condition>]

Purpose:

To summarize records by key value, sum specified numeric fields, and copy summary records to a second database file.

Arguments:

<key exp> defines the group of records as encountered that produce a new record in the target database file. To make the summarizing operation accurate, the source database file should be INDEXed or SORTed on the TOTAL key expression.

<file> is the name of the target file to copy the summarized records. Unless otherwise specified, TOTAL assumes a (.dbf) extension.

Options:

Fields: The FIELDS clause specifies the list of numeric fields to TOTAL. If the FIELDS clause is not specified, no numeric fields are totalled. Instead each numeric field in the target file contains the value for the first record matching the key expression.

Scope: The <scope> is the portion of the current database file to TOTAL. The default is ALL.

Condition: The FOR clause specifies the conditional set of records to TOTAL within the given scope. The WHILE clause specifies the set of records meeting the condition from the current record until the condition fails.

Usage:

In Clipper, TOTAL has two modes depending on whether you specify a FIELDS clause or not. If the FIELDS clause is specified, TOTAL sums the specified numeric fields to the target database file grouped by the specified key expression. If, however, the FIELDS clause is not specified, TOTAL copies only records with unique keys to the target database file. With this formulation, you can eliminate records with duplicate key values.

The structure of the target database file Clipper creates is identical to the source except that memo fields are not copied. Note that in order to successfully TOTAL numeric fields, the

source numeric fields must be large enough to hold the largest total possible for that numeric field.

Example:

```
USE Sales
? LASTREC()                          && Result: 84
TOTAL ON Branch + Salesman;
      FIELDS Amount;
      TO Summary

USE Summary
? LASTREC()                          && Result: 5
```

Library: CLIPPER.LIB

See also: AVERAGE, SUM

TYPE

Syntax: TYPE <file>.<ext> [TO PRINT] [TO FILE <file>]

Purpose: To display the contents of a text file to the screen, optionally echoing the display to the printer and/or another text file.

Argument: <file>.<ext> is the name of the file including extension to display to the screen.

Options: **Print:** The TO PRINT clause echoes the display to the printer.

File: The TO FILE clause echoes the display to the specified <file>. If no extension is specified, (.txt) is added.

Usage: To pause output, use Ctrl-S. Note that you cannot interrupt the listing with Esc.

Example:

```
TYPE Main.prg TO PRINT
```

Library: CLIPPER.LIB

See also: COPY FILE

UNLOCK

Syntax: UNLOCK [ALL]

Purpose: To release a file or record locks set by the current user.

Option: **All:** The ALL clause releases all current locks in all work areas. If it not specified locks in the current work area are released.

Example: See Chapter 10, *Using Clipper With A Local Area Network*, for additional information.

Library: CLIPPER.LIB

See also: SET EXCLUSIVE, USE...EXCLUSIVE, FLOCK(), RLOCK()

UPDATE

Syntax:

UPDATE ON <key exp> FROM <alias> REPLACE <field1> WITH <exp1> [,<field2> WITH <exp2>]... [RANDOM]

Purpose:

To update the current database file from another database file based on a one-to-one or one-to-many relation.

Arguments:

<key exp> is an expression used that defines matching records in the source work area.

<alias> is the alias of the source work area used to update records in the current work area.

<field1...fieldN> are the fields in the current work area to replace with new values.

<exp1...expN> identifies the values to replace into the target fields. Fields from the source work area must be referenced with the alias of the respective work area.

Option:

Random: The RANDOM clause causes the entire source database file to be read allowing the source database file to be in any order. If this option is specified, the target database file must be indexed on the <key exp>.

Usage:

UPDATE matches records in the target work area using the key expression as the argument of a lookup into the source work area. To obtain an accurate UPDATE, both database files must be sorted or indexed on the key expression. If the source database file is not sorted or indexed, use the RANDOM clause.

Relations between work areas: UPDATE supports both one-to-one and one-to-many relations between the target and the source work areas. It does not, however, support many-to-many or many-to-one relations. All source records matching the <key exp> UPDATE only the first instance of the matching target record.

Types of replacement: When you UPDATE, the REPLACE expression determines the type of UPDATE made to a target

field. If the UPDATE to the field adds to the target field, the formulation is:

```
REPLACE <target field> WITH <target field> + <expN>
```

Likewise, if the UPDATE subtracts from the target field, the formulation is:

```
REPLACE <target field> WITH <target field> - <expN>
```

Deleted records: When DELETED is ON, all target records are UPDATEd including deleted ones. Deleted source records, however, are ignored.

Network:

In order to UPDATE in a network environment, the target database file must be locked with FLOCK() or USEed EXCLUSIVEly. The source file may either be USEd EXCLUSIVEly or shared.

Examples:

This example UPDATEs a Customers database file with outstanding invoice amounts:

```
USE Customers INDEX Customers
SELECT B
USE Invoices
SELECT Customers
*
UPDATE FROM Invoices RANDOM;
    ON Last;
    REPLACE Owed WITH;
    Owed + Invoices->Amount
```

Library:

CLIPPER.LIB

See also:

APPEND, REPLACE, SET UNIQUE

USE

Syntax:

USE [<file>/(<expC>)] [INDEX <file list>/(<expC>) [,(<expC>)]...] [EXCLUSIVE] [ALIAS <alias>/(<expC>)]

Purpose:

To open an existing database file (.dbf), its associated memo file (.dbt), and optionally associated index files in the currently selected work area.

Argument:

<file> is the name of the database file to open.

Options:

Index: The INDEX <file list> clause specifies the name(s) of up to 15 index files to open in the current work area.

Alias: The <alias> is the name to associate with the work area when the database file is opened. If this clause is not specified, the alias defaults to the database filename.

Exclusive: The EXCLUSIVE clause opens the database file for non-shared use in a network environment. All other users are denied access until the database file is CLOSEd.

Usage:

When a database file is opened, the record pointer is positioned at the first logical record in the file (record 1 if there is no index file specified).

USE without an argument CLOSEs the active database file, associated index, and memo files in the current work area.

Example:

```
USE Accounts EXCLUSIVE
IF NETERR()                    && USE not successful.
   ? "Accounts file not available"
ELSE
   SET INDEX TO Acct_ID
ENDIF
```

See Chapter 10, *Using Clipper with a Local Area Network*, for additional information.

Library:

CLIPPER.LIB

See also:

CLOSE, INDEX, SELECT, SET INDEX

WAIT

Syntax:

WAIT [<prompt>] [TO <memvarC>]

Purpose:

To pause program execution after displaying a prompt and then waiting until a key is pressed.

Options:

Prompt: The <prompt> is a character string WAIT displays if specified. If no <prompt> is specified, the default prompt is "Press any key to continue..."

Memory variable: The TO <memvarC> clause creates the specified memory variable that contains the character entered.

Usage:

WAIT returns the character entered into the specified character memory variable. If a non-printable character is entered, WAIT returns zero to the character variable. Function keys are ignored unless assigned with SET FUNCTION or SET KEY.

Example:

```
WAIT "Press a key..." TO key
```

Library:

CLIPPER.LIB

See also:

ACCEPT, INPUT, INKEY()

ZAP

Syntax: ZAP

Purpose: To remove all records from the currently selected database file.

Usage: ZAP is the same as DELETE ALL followed by PACK, but is much faster. Index or memo files in USE are also ZAPped.

Example:

```
USE Sales
? LASTREC()                          && Result: 84
COPY TO Archive
ZAP
? LASTREC()                          && Result: 0
```

Network: To ZAP in a network environment, the current database file must be USEed EXCLUSIVEly.

Library: CLIPPER.LIB

See also: CLEAR, DELETE, PACK

6 Clipper Functions

Chapter 6 contains a summarized list and detailed descriptions of the Clipper functions. Topics covered include:

- A summary of all functions used in Clipper, including syntax, arguments, return values, usage discussion, and examples.

SUMMARY OF ALL CLIPPER FUNCTIONS

Clipper functions are contained in one of two libraries (CLIPPER.LIB and EXTEND.LIB). Note the libraries containing the functions you require. They must be specified when linking your application. For more information on linking and compiling, see Chapter 7, *"Compiling and Linking Programs."*

Function Name, Syntax And Purpose

ABS(<expN>)

Returns the absolute value of a numeric expression.

ACHOICE(<expN1>, <expN2>, <expN3>, <expN4>, <array1> [,<array2> [,<expC> [,<expN5> [,<expN6>]]]])

Executes a pop-up menu using an array of character strings as choices and returns the selection as a numeric value.

ACOPY(<array1>, <array2> [,<expN1> [,<expN2> [,<expN3>]]])

Copies elements from one array to another.

ADEL(<array>, <expN>)

Deletes an array element.

ADIR(<directory skeleton> [,<array1> [,<array2> [,<array3> [,<array4> [, <array5>]]]]])

Fills a series of arrays with file information from the disk directory.

ADEL(<array>, <expN>)

Deletes an array element.

AFIELDS([<array1> [,<array2> [,<array3>[,<array4>]]]])

Fills a series of arrays with field definition information.

AFILL(<array>, <exp> [,<expN1> [,<expN2>]])

Fills an array with one value.

AINS(<array>, <expN>)

Inserts a new position into an array.

ALIAS([<expN>])

Returns the alias of a work area.

ASC(<expC>)

Returns the ASCII code value of the left most character in a character string as a numeric.

ASCAN(<array>, <exp> [,<expN1> [,<expN2>]])

Searches for a specific value within an array.

ASORT(<array>, [,<expN1> [,<expN2>]])

To sort an array in ascending order.

AT(<expC1>, <expC2>)

Returns a number giving the starting position of a character string within another character string.

BOF()

Indicates the beginning of a file.

CDOW(<expD>)

Returns the day of the week from a date value.

CHR(<expN>)

Returns a character for the specified ASCII code.

CMONTH(<expD>)

Returns the name of the month from a date value.

COL()

Returns the current column position of the cursor.

CTOD(<expC>)

Converts a date that has been stored or entered as a character string into a date value.

DATE()

Returns the system date in the form "mm/dd/yy."

DAY(<expD>)

Returns the numeric value of the day of the month from a date value.

DBEDIT([<expN1> [,<expN2> [,<expN3> [,<expN4>]]]] [,<array1>] [,<expC>] [,<array2>] [,<array3>] [,<array4>] [,<array5>] [,<array6>] [,<array7>])

Displays and edits records from one or more work areas using a browse-style editor that executes within a defined window area.

DELETED()

Returns the deletion status of the current record as a logical value.

DOW(<expD>)

Returns a number that represents the day of the week from a date value.

DTOC(<expD>)

Converts a date value to a character string.

DTOS(<expD>)

Returns a character string in the form "yyyymmdd" in order to INDEX ON a date and a character expression.

EMPTY(<exp>)

Returns true (.T.) if <exp> is blank.

EOF()

Returns true (.T.) if the end of file is reached.

EXP(<expN>)

Calculates e^x where e is the base of natural logarithms and x is the numeric argument.

FCLOSE(<expN>)

Closes the DOS file that corresponds to the specified handle.

FCOUNT()

Returns the number of fields in the current database file.

FCREATE(<expC> [,<expN>])

Creates a new DOS file or truncates an existing file to zero length.

FERROR()

Returns the DOS error number after a file function terminates.

FIELD/FIELDNAME(<expN>)

Returns the name of the specified field in the current work area.

FILE(<file>)

Returns true (.T.) if a specified filename exists.

FLOCK()

Locks a database file including the memo and index files in a shared environment, preventing two users from writing to it at the same time.

FOPEN(<expC> [,<expN>])

Opens a DOS file.

FOUND()

Returns true (.T.) if the previous SEEK, FIND, LOCATE or CONTINUE command was successful.

FREAD(<expN1>, @<memvarC>, <expN2>)

Reads characters from a DOS file into a character memory variable.

FREADSTR(<expN1>, <expN2>)

Reads characters from a DOS file.

FSEEK(<expN1>, <expN2> [,<expN3>])

Moves the file pointer to a new position in a DOS file.

FWRITE(<expN1>, <memvarC> [,<expN2>])

Writes a buffer variable to a specified DOS file.

HARDCR(<expC>)

Replaces all soft carriage returns with hard carriage returns within a character expression.

IF/IIF(<expL>, <exp1>, <exp2>)

Provides for conditional processing of expressions.

INDEXEXT()

Returns "NTX" or "NDX" to indicate the type of index that is currently active.

INDEXKEY(<expN>)

Returns the key expression of a specified index.

INDEXORD()

Returns the controlling index as a numeric value that points into the list of indexes specified by SET INDEX TO or USE...INDEX commands.

INKEY([<expN>])

Returns the numeric value of the ASCII code for the key pressed.

INT(<expN>)

Converts any numeric expression into an integer by truncating all digits to the right of the decimal.

ISALPHA(<expC>)

Returns true (.T.) if the first character in <expC> is alphabetic.

ISCOLOR()

Returns true (.T.) if a color display is installed and false (.F.) if a monochrome display is installed.

LASTKEY()

Returns the numeric value of the ASCII code for the last key pressed.

LASTREC()/RECCOUNT()

Returns the number of records in the current work area.

LEN(<expC>/<array>)

Returns the number of characters in a specified character string or the number of elements in a specified array.

LOG(<expN>)

Returns the natural logarithm of a given number.

LOWER(<expC>)

Converts upper case characters to lower case.

LTRIM<(expC>)

Removes leading blanks from a character expression.

MAX(<expN1>/<expD1>,<expN2>/<expD2>)

Returns the greater of two numeric or date expressions.

MEMOEDIT(<expC> [,<expN1>, <expN2>, <expN3>, <expN4>] [,<expL>])

Displays and edits memo fields and long strings.

MEMOLINE(<expC>, <expN1>, <expN2>)

Returns a formatted line from a character expression or memo field.

MEMOREAD(<expC>)

Returns the specified disk file as a character string.

MEMORY()

Returns the available free pool memory.

MEMOTRAN(<expC1> [,<expC2> [,<expC3>]])

Returns a character string with the carriage return and line feed characters replaced.

MEMOWRIT(<expC1>, <expC2>)

Writes a character string to a specified disk file and returns true (.T.) if successful.

MIN(<expN1>/<expD1>,<expN2>/<expD2>)

Returns the lesser value of two numeric or date expressions.

MLCOUNT(<expC>, <expN>)

Returns the number of lines in a character expression or memo field.

MONTH(<expD>)

Returns a number representing the month from a date variable.

NETERR()

Returns true (.T.) if a USE, USE...EXCLUSIVE, APPEND BLANK, FLOCK() or RLOCK() fails in a network environment.

NETNAME()

Returns the text of the computer name.

NEXTKEY()

Reads the next keystroke without removing it from the keyboard buffer.

PCOL()

Returns the current column position on the printer.

PCOUNT()

Returns the number of actual parameters passed to a procedure or user-defined function.

PROCLINE()

Returns the source code line number of the current program being executed.

PROCNAME()

Returns the name of the current program or procedure being executed.

PROW()

Returns the current row on the printer.

RAT(<expC1>, <expC2>)

Searches a character string for the last instance of a specified substring and returns the starting position as a numeric value.

READVAR()

Returns the name of the current GET/MENU variable or a null string if none is pending.

RECNO()

Returns the current record number of the current work area.

REPLICATE(<expC>, <expN>)

Repeats a character expression a specified number of times.

RLOCK()/LOCK()

Attempts to lock the current record.

ROUND(<expN1>, <expN2>)

Returns rounded numbers to a specified number of decimal places.

ROW()

Returns the current screen row location of the cursor.

SCROLL(<expN1>, <expN2>, <expN3>, <expN4>, <expN5>)

Uses window coordinates to designate a section of the screen that can be scrolled up and down or blanked out.

SECONDS()

Returns the system time as <seconds>.<hundredths>.

SELECT()

Returns the number of the current work area.

SETPRC(<expN1>, <expN2>)

SETS the internal PROW() and PCOL() functions to the specified values.

SPACE(<expN>)

Creates a character string consisting of a specified number of spaces.

SQRT(<expN>)

Returns the square root of a given positive number.

STRTRAN(<expC1>, <expC2> [,<expC3>] [,<expN1>] [,<expN2>])

Searches and replaces within a character string.

SUBSTR(<expC>, <expN1> [,<expN2>])

Extracts a specified part of a character string.

TIME()

Returns the system time in the format "hh:mm:ss."

TRANSFORM(<exp>, <expC>)

Returns a character string with the specified picture format.

TRIM(<expC>)/RTRIM()

Removes trailing blanks from a character expression.

TYPE("<expC>")

Returns the data type of a memory variable, expression, or field.

UPDATED()

Returns true (.T.) if the last READ changed any data in the associated GETs.

UPPER(<expC>)

Converts lower case characters to upper case.

VAL(<expC>)

Converts a character string to a numeric value.

WORD()

Converts numeric arguments of the CALL command from type DOUBLE to type INT.

YEAR(<expD>)

Returns the complete numeric value of the year from a date value.

ABS()

Syntax:

ABS(<expN>)

Purpose:

To evaluate a numeric expression and return the absolute value.

Argument:

<expN> is a numeric expression to evaluate.

Returns:

A numeric value.

ABS() returns a positive number or zero for any numeric expression argument, whether positive or negative.

Usage:

ABS() is used wherever you need to know the magnitude of a numeric expression. This allows you to evaluate the difference between two numbers when you are not concerned with the sign of the result.

Examples:

```
a = 100
b = 150
? a - b                    && Result: -50
? ABS(a - b)               && Result: 50
? ABS(-12)                 && Result: 12
? ABS(0)                   && Result: 0
```

Library:

CLIPPER.LIB

ACHOICE()

Syntax:

ACHOICE(<expN1>, <expN2>, <expN3>, <expN4>, <array1>
[,<array2> [,<expC> [,<expN5> [,<expN6>]]]])

Purpose:

To execute a pop-up menu using an array of character strings as choices.

Arguments:

<expN1...expN4> are the top, left, bottom and right window coordinates respectively.

<array1> is an array of character strings to display as menu choices.

<array2> is a parallel array of logical values, one for each menu choice. If an element is false (.F.), the parallel menu choice is not available. You can, however, specify this argument as a single logical value and the menu is treated as if you specified an array filled with that value. This is valuable when you want all choices to be either available or unavailable.

<expC> is a user-defined function that executes generally when a key exception is pressed. Be sure to specify the function name without the parenthetical suffix or any arguments. Note that the behavior of ACHOICE() is affected by the presence of this argument. Refer to the discussion below for further information.

<expN5> is the initial choice element. If not specified, the default initial choice is element one in <array1>.

<expN6> is the initial relative window row. If not specified, the initial relative window row is first choice in the window, which is position zero.

Returns:

A numeric value.

ACHOICE() returns the position of the menu choice in the array of choices. Making a selection terminates the menu and returns the current element position in <array1>. Aborting the selection terminates the menu and returns zero.

Usage:

ACHOICE() has two modes depending on whether or not you specify the user-defined function argument (<expC>). If you do not specify a user function, ACHOICE() displays the list of choices within the given screen coordinates. It then executes the following actions when you press the corresponding keys.

Table 6-1 ACHOICE() Active Keys - No User Function

Key	Action
Uparrow	Up one element
Dnarrow	Down one element
Home	First element
End	Last element
PgUp	Up the number of elements defined for the menu window to the same relative row position
PgDn	Down the number of elements defined for the menu window to the same relative row position
Ctrl-PgDn	First element
Ctrl-PgUp	Last element
Return	Select element, returning position
Esc	Abort selection, returning zero
Leftarrow	Abort selection, returning zero
Rightarrow	Abort selection, returning zero
First letter	Next element with same first letter

If the number of choices in <array1> exceeds the number of rows in the menu window, the choices scroll when you attempt to cursor beyond the top or bottom of the menu window. Note that the highlight does not wrap when you reach the top or bottom of the list of choices. Pressing the first letter does, however, wrap the highlight within the set of choices whose first letter matches the key you are pressing.

The User Function: If you specify a user function, the behavior of ACHOICE() changes to some degree. Primarily, fewer keys are now automatically executed by ACHOICE() and control passes to the user function where you can process the key pressed and execute an appropriate action. The following are keys that ACHOICE() executes when you specify a user function. These keys generate an idle (mode 0) while all other keys generate a keyboard exception (mode 3).

Table 6-2 ACHOICE() Active Keys - User Function

Key	Action
Uparrow	Up one element
Dnarrow	Down one element
PgUp	Up the number of elements defined for the menu window to the same relative row position
PgDn	Down the number of elements defined for the menu window to the same relative row position
Ctrl-PgDn	First element
Ctrl-PgUp	Last element

Note that Home, End, Return, and Esc now generate a keyboard exception (mode 3).

When ACHOICE() executes the user function, it automatically passes the following three parameters: mode, current element in the array of choices, and the relative position within the menu window. The mode indicates the current state of ACHOICE() depending on the key pressed and action taken by ACHOICE() prior to executing the user function. The mode parameter has the following possible values:

Table 6-3 ACHOICE() Modes

Mode	Description
0	Idle
1	Cursor past top of list
2	Cursor past end of list
3	Keystroke exception
4	No item selectable

After your user function has performed whatever operations are appropriate to the ACHOICE() mode or the last key pressed, you RETURN a value instructing ACHOICE() what operation to perform next. The following table summarizes the possible return values:

Table 6-4 Return Values to ACHOICE() from User Function

Value	Action
0	Abort selection, return zero
1	Make selection, return cursor element
2	Continue selection process
3	Go to the next element whose first character matches the last key pressed

Color: Choices are displayed in standard color, the highlight is in enhanced color, and the unavailable choices are displayed in the unselected color. For example, the following color statement:

```
SET COLOR TO W+/N,BG+/B,,,W/N
```

displays a menu that is bright white on black, the highlight is bright cyan on blue, and the unavailable menu choices are dim white on black.

If you are executing ACHOICE() with all choices unavailable in order to have a display-only list, assign the unselected setting to the same setting as the standard.

Nesting: You can call multiple copies of ACHOICE() within each copy of ACHOICE() allowing you to create nested or hierarchical menus.

Example:

```
DECLARE pad[3]
pad[1] = "one  "
pad[2] = "two  "
pad[3] = "three"
menuchoice = ACHOICE(10, 10, 12, 15, pad)
```

Library: EXTEND.LIB

See also: @...PROMPT, MENU TO, SET MESSAGE, DBEDIT()

ACOPY()

Syntax:

ACOPY(<array1>, <array2> [,<expN1> [,<expN2> [,<expN3>]]])

Purpose:

To copy elements from one array to another.

Arguments:

<array1> is the source array.

<array2> is the target array.

<expN1> is the starting element position in the source array.

<expN2> is the number of elements to copy from the source array beginning with <expN1>.

<expN3> is the starting element position in the target array to begin the copy.

Returns:

There is no return value.

Example:

```
DECLARE one[5], two[5]
ADIR("*.*", one)
ADIR("*.prg", two)
fun = ACOPY(one, two, 1, 2)
FOR i = 1 to 5
    ? one[i], two[i]
NEXT
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ADEL(), ADIR(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT(), LEN()

ADEL()

Syntax: ADEL(<array>, <expN>)

Purpose: To delete an array element.

Arguments: <array> is the name of the array to delete an element from.

<expN> is the position of the element to delete.

Returns: There is no return value.

Usage: The contents of the indicated array position are lost and all elements from that position to the end of the array are shifted up one element. The last position in the array then becomes undefined until a new value is assigned to it.

Example:

```
DECLARE array[3]
array[1] = 1
array[2] = 2
array[3] = 3
? array[2]                    && Result: 2
ADEL(array, 2)
? array[2]                    && Result: 3
```

Library: EXTEND.LIB

See also: ACHOICE(), ACOPY(), ADEL(), ADIR(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT(), LEN()

ADIR()

Syntax:

ADIR(<directory skeleton> [,<array1> [,<array2> [,<array3> [,<array4> [, <array5>]]]]])

Purpose:

To fill a series of arrays with directory information including filenames and/or return the number of files matching a skeleton.

Arguments:

<directory skeleton> is a wildcard pattern of files. The standard wildcard characters (* and ?) are supported. The default is "*.*"

<array1> is the name of the array to fill with the filenames matching the directory skeleton. Each element is character type.

<array2> is the array to fill with the sizes of the corresponding files in <array1>. Each element is numeric type.

<array3> is the array to fill with the dates of the corresponding files in <array1>. Each element is date type.

<array4> is the array to fill with the times of the corresponding files in <array1>. Each element is character type.

<array5> is the array to fill with attributes of the corresponding files in <array1>. Each element is character type. The possible values returned are as follows:

Table 6-5 Attributes of Files Returned by ADIR()

Symbol	Description
R	Read only
H	Hidden
S	System
D	Directory
A	Archive

Note that if <array5> is specified, hidden, system, and directory files are included as well as normal files. If <array5> is not specified only normal files are included.

Returns:

An integer numeric value.

ADIR() returns the number of files matching the specified directory skeleton. If the optional array name is included, elements of the array are filled with one filename per element until all matching files have been found or all of the elements have been used.

Usage:

ADIR() is useful as a tool for building file selectors and directory maintenance routines. To create a file list array, you must DECLARE an array and then fill it with the necessary filenames. The best way to do this is to first use ADIR() as the number of elements argument in the DECLARE statement. For example:

```
DECLARE dbf_files[ADIR("*.dbf")]
```

Then sometime later, fill the array using the directory skeleton.

To fill arrays for some attributes while ignoring others, pass a dummy variable. For example, to obtain only the filenames and corresponding file dates:

```
dbf_count = ADIR("*.dbf")
DECLARE dbf_files[dbf_count], dbf_date[dbf_count]
dummy = ""
ADIR(dbf_file, dummy, dbf_date)
```

Example:

```
DECLARE aray_dir[ADIR("*.txt")]
ADIR("*.txt", aray_dir)
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), AFIELDS(), AFILL(), AINS(), ASCAN(), ASORT(), LEN()

AFIELDS()

Syntax:

AFIELDS([<array1> [,<array2> [, <array3> [,<array4>]]]])

Purpose:

To fill a series of arrays with fieldnames, field types, field lengths, and field decimals.

Arguments:

<array1> is the array to fill with fieldnames. Each element is character type.

<array2> is the array to fill with the type of fields in <array1>. Each element is character type.

<array3> is the array to fill with the widths of fields in <array1>. Each element is numeric type.

<array4> is the array to fill with the number of decimals defined for fields in <array1>. Each element is numeric type. If the field type is not numeric, the element is zero.

Returns:

An integer numeric value.

AFIELDS() returns the number of fields or the length of the shortest array argument, whichever is less. If no parameters are specified, AFIELDS() returns zero.

Usage:

AFIELDS() fills a series of arrays with the attributes of fields from the current work area and returns the number of elements filled with field information. If there is no database file in USE, AFIELDS() returns zero. The arrays passed correspond to fieldname, field type, field length, and field decimals if the field is numeric.

To fill arrays for some attributes while ignoring others, pass a dummy variable. For example, to obtain the fieldnames and corresponding field lengths only:

```
DECLARE fname[FCOUNT()], ftype[FCOUNT()]
dummy = ""
AFIELDS(fname, dummy, ftype)
```

Examples:

The following demonstrates how to use AFIELDS() and ACHOICE() to create and select from a fields list:

```
CLEAR
USE Sales
*
DECLARE fname[FCOUNT()]
AFIELDS(fname)
@ 1,0 TO 10,10 DOUBLE
fld = ACHOICE(2, 1, 9, 9, fname)
@ 12, 0 SAY IF(fld  0, fname[fld], "None selected")
RETURN
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFILL(), AINS(), ASCAN(), ASORT(), FIELD(), LEN()

AFILL()

Syntax:

AFILL(<array>, <exp> [,<expN1> [,<expN2>]])

Purpose:

To fill an array with a chosen value.

Arguments:

<array> is the array to fill.

<exp> is the value to place in each array element. It can be an expression of any data type.

<expN1> is the position of the first element to fill. This argument is optional and defaults to 1.

<expN2> is the number of elements to fill starting with element <expN1>. This argument is optional and defaults to all elements from the starting element to the end of the array.

Returns:

There is no return value.

Usage:

Note that AFILL() does not work the same as similar functions in spreadsheet languages. Specifically, there is no provision for incrementing values in the range specified.

Examples:

```
DECLARE alogic[15]
AFILL(alogic, .F.)
AFILL(alogic, .T., 5, 10)
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFIELDS(), AINS(), ASCAN(), ASORT(), LEN()

AINS()

Syntax:

AINS(<array>, <expN>)

Purpose:

To insert an undefined element into an array.

Arguments:

<array> is the array to insert a new element into.

<expN> is the position to insert the new element.

Returns:

There is no return value.

Usage:

The newly inserted position remains undefined until a new value is assigned to it. After the insertion, all elements after the new element are shifted down one position and the last array element is lost.

Example:

```
DECLARE array[3]
array[1] = 1
array[2] = 2
array[3] = 3
? array[2]                    && Result: 2
AINS(array, 2)
? array[3]                    && Result: 2
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFIELDS(), AFILL(), ASCAN(), ASORT(), LEN()

ALLTRIM()

Syntax:

ALLTRIM(<expC>)

Purpose:

To remove leading and trailing spaces.

Argument:

<expC> is the character string to trim.

Returns:

A character string.

ALLTRIM() returns <expC> without leading or trailing blanks and is the equivalent of LTRIM(RTRIM(<expC>)).

Examples:

```
charvar = SPACE(10) + "string" + SPACE(10)
? LEN(charvar)                    && Result: 26
? LEN(ALLTRIM(charvar))           && Result: 6
```

Library:

EXTEND.LIB

See also:

LTRIM(), RTRIM()

ALTD()

Syntax:

ALTD([<expN>])

Purpose:

To execute the Debugger or enable/disable the use of Alt-D to invoke it.

Argument:

<expN> sets the Alt-D invocation state as follows:

Table 6-5.1 ALTD() Arguments

Argument	Action
None	Invokes Debugger (last screen)
0	Disables Alt-D
1	Enables Alt-D
2	Invokes Debugger (View Privates)

Returns:

There is no return value.

Usage:

If there is no argument specified, the Debugger is invoked displaying the last screen displayed. Specifying two as the argument invokes the Debugger displaying the Variables:View Privates screen.

Note that ALTD(), when specified with no argument or two as the argument, sets the invocation state for subsequent invocations using the Alt-D key.

Library:

CLIPPER.LIB

See also:

SET ESCAPE, SETCANCEL()

ALIAS()

Syntax: ALIAS([<expN>])

Purpose: To obtain the alias name of the specified work area.

Argument: <expN> is the number of specified work area.

Returns: A character string.

If no argument is specified, the alias of the current work area is returned. If a numeric argument is specified, the alias of that area is returned. If there is no database file in USE, ALIAS() returns a null string ("").

Example:

```
USE Sales
SELECT 2
USE Client
? ALIAS(1), ALIAS()     && Result: "Sales Client"
```

Library: CLIPPER.LIB

See also: SELECT, USE, SELECT()

ASC()

Syntax: ASC(<expC>)

Purpose: To return the ASCII or IBM Extended Character Set value of the left most character in a character expression.

Argument: <expC> is the character expression to convert to a number.

Returns: An integer numeric value in the range of 0 to 255.

Usage: ASC() is used primarily in expressions where you need to perform numeric calculations on the ASCII value of a character.

Examples:

```
? ASC("A")                    && Result: 65
? ASC("Apple")                && Result: 65
? ASC("a")                    && Result: 97
? ASC("Z") - ASC("A")         && Result: 25
? ASC("")                     && Result: 0
```

Library: CLIPPER.LIB

See also: CHR(), INKEY()

ASCAN()

Syntax:

ASCAN(<array>, <exp> [,<expN1> [,<expN2>]])

Purpose:

To scan an array for a specific value.

Arguments:

<array> is the array to scan.

<exp> is the value to scan for. This can be a valid expression of any data type.

<expN1> is the starting element of the scan. This argument is optional and defaults to one if not specified.

<expN2> is the number of elements to scan from the starting position. This argument is optional and defaults to all elements from the starting element to the end of the array.

Returns:

An integer numeric value.

ASCAN() returns the element position containing the matching value. If no value is found, ASCAN() returns zero.

Usage:

ASCAN() works the same as SEEK and FIND in the way it performs a search. The <exp> is tested against the target array element beginning with the left most character in the target element and proceeds until there are no more characters left in the <exp>. If there is no match, ASCAN() proceeds to the next element in the array. Note that ASCAN() is also sensitive to the status of EXACT. If EXACT is ON, the target array element must match the result of <exp> character for character.

Example:

```
DECLARE dir_list[ADIR("*.txt")]
ADIR("*.txt", dir_list)
*
ptr = ASCAN(dir_list, "TEMP.TXT")
IF ptr > 0
   dir_list[ptr] = "NEWTEMP.TXT"
ELSE
   ? "No match was found"
ENDIF
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFIELDS(), AFILL(), AINS(), ASORT(), LEN()

ASORT()

Syntax:

ASORT(<array>, [,<expN1> [,<expN2>]])

Purpose:

To sort the contents of an array in ascending order.

Arguments:

<array> is the array to sort.

<expN1> is the first element of the sort. If omitted, the sort begins with position 1.

<expN2> is the number of elements to sort. If you omit this argument, the sort operation proceeds from the beginning element position to the end of the array.

Returns:

There is no return value.

Usage:

All elements in the range of the array being sorted must be the same data type.

Example:

```
DECLARE testarray[3]
testarray[1] = "AA"
testarray[2] = "CC"
testarray[3] = "BB"
*
ASORT(testarray)
*
? testarray[1]                    && Result: "AA"
? testarray[2]                    && Result: "BB"
? testarray[3]                    && Result: "CC"
```

Library:

EXTEND.LIB

See also:

ACHOICE(), ACOPY(), ADEL(), ADIR(), AFIELDS(), AFILL(), AINS(), ASCAN(), LEN()

BIN2I()

Syntax:

BIN2I(<expC>)

Purpose:

To convert a character string formatted as a 16-bit signed integer to a Clipper numeric value.

Argument:

<expC> is a two-byte string in the form of a 16-bit signed integer number. If more than two characters are specified, the remaining are ignored.

Returns:

An integer numeric value.

Usage:

BIN2I() is used in combination with FREAD() or FREADSTR() to convert a two-byte character string formatted as a signed integer to a Clipper numeric. This is most useful when you are reading foreign file types and want to read numeric data formatted in its native form.

Example:

This example opens a database file using low-level file functions and reads the date of last update (bytes 1-3). The result is the same as with LUPDATE().

```
handle = FOPEN("Sales.dbf")
*
* Point to byte 1 in the file.
FSEEK(handle, 1, 0)
*
* Read date of last update.
year  = BIN2I(FREADSTR(handle, 1))
month = BIN2I(FREADSTR(handle, 1))
day   = BIN2I(FREADSTR(handle, 1))
*
? LTRIM(STR(month)),;          && Result: 9  1  87
    LTRIM(STR(day)), LTRIM(STR(year))
FCLOSE(handle)
```

Library:

EXTEND.LIB

Source:

EXAMPLEA.ASM

See also:

BIN2L(), BIN2W(), I2BIN(), L2BIN(), FOPEN(), FREAD(), FREADSTR()

BIN2L()

Syntax: BIN2L(<expC>)

Purpose: To convert a character string formatted as a 32-bit signed long integer to a Clipper numeric value.

Argument: <expC> is a four-byte character string in the form of a 32-bit signed integer number. If more than four characters are specified, the remaining are ignored.

Returns: An integer numeric value.

Usage: BIN2W() is used in combination with FREAD() or FREADSTR() to convert a four-byte character string formatted as a signed long integer to a Clipper numeric. This is most useful when you are reading foreign file types and want to read numeric data formatted in its native form.

Example: This example opens a database file using low-level file functions and reads the number of records (bytes 4-7). The result is the same as with LASTREC().

```
handle = FOPEN("Sales.dbf")
num_recs = SPACE(4)
*
* Point to byte 4.
FSEEK(handle, 4, 0)
*
* Read the number of records.
num_recs = SPACE(4)
FREAD(handle, @num_recs, 4)
*
? LTRIM(STR(BIN2L(num_recs)))     && Result: 84
FCLOSE(handle)
```

Library: EXTEND.LIB

Source: EXAMPLEA.ASM

See also: BIN2I(), BIN2W(), I2BIN(), L2BIN(), FOPEN(), FCLOSE()

BIN2W()

Syntax:

BIN2W(<expC>)

Purpose:

To convert a character string formatted as a 16-bit unsigned integer to a Clipper numeric value.

Argument:

<expC> is a two-byte character string in the form of a 16-bit unsigned or long integer number.

Returns:

An integer numeric value.

Usage:

BIN2W() is used in combination with FREAD() or FREADSTR() to convert a two-byte character string formatted as an unsigned integer to a Clipper numeric. This is most useful when you are reading foreign file types and you want to read numeric data formatted in its native form.

Example:

This example opens a database file using low-level file functions and reads the number of bytes per record (bytes 10-11). The result is the same as with RECSIZE().

```
handle = FOPEN("Sales.dbf")
*
* Point to byte 10, the first record size byte.
FSEEK(handle, 10, 0)
*
* Read record size.
rec_size = SPACE(2)
FREAD(handle, @rec_size, 2)
*
? LTRIM(STR(BIN2W(rec_size)))     && Result: 124
FCLOSE(handle)
```

Library:

EXTEND.LIB

Source:

EXAMPLEA.ASM

See also:

BIN2I(), BIN2L(), I2BIN(), L2BIN(), FOPEN(), FREAD(), FREADSTR()

AT()

Syntax:

AT(<expC1>, <expC2>)

Purpose:

To search a character string for the first instance of a specified substring and return the starting position as a numeric value.

Arguments:

<expC1> is the character string to locate.

<expC2> is the character string to be searched.

Returns:

An integer numeric value.

If the substring is contained within the target expression, AT() returns the starting character position of the substring. If the substring is not found, AT() returns zero.

Usage:

AT() is a general purpose character string manipulation function. Generally, you use AT() to determine the location of the first instance of a substring within a string when you need it as a numeric value. If you only need to know whether a string is contained within another, use the $ operator.

Examples:

```
? AT("a", "abcde")          && Result: 1
? AT("bcd", "abcde")        && Result: 2
? AT("a", "bcde")           && Result: 0
```

Library:

CLIPPER.LIB

See also:

RAT(), STRTRAN(), SUBSTR(), $

BOF()

Syntax:

BOF()

Purpose:

To determine if an attempt has been made to move the record pointer past the beginning of the current database file.

Returns:

A logical value.

BOF() returns true (.T.) only when you attempt to move the record pointer before the first logical record in the current database file. When this happens, the record pointer is positioned to the first logical record. If the current database file contains no records, both BOF() and EOF() return true (.T.).

Note that SKIP is the only record movement command that can set BOF() true (.T.).

Usage:

BOF() is generally used as a boundary condition test in applications where you need to move the record pointer backwards through the database file. A simple application is a descending order record list without a descending order index file. A more sophisticated application is a screen paging routine that pages forward or backward through the current database file based on the key you press. You would use BOF() here to test when you have reached top-of-file.

Examples:

```
USE Sales
GO TOP
? RECNO()                && Result: 1
? BOF()                  && Result: .F.
*
SKIP -1
? RECNO()                && Result: 1
? BOF()                  && Result: .T.
```

Library:

CLIPPER.LIB

See also:

SKIP, EOF()

CDOW()

Syntax:

CDOW(<expD>)

Purpose:

To convert a date value to a string representing the name of the day of the week.

Argument:

<expD> is the date value to convert.

Returns:

A character string.

CDOW() returns the name of the day of the week with the first letter in upper case and the rest of the string in lower case. The maximum return value length is nine characters for Wednesday. A null date value returns a null string ("").

Usage:

CDOW() is useful alone or as part of date formatting expressions for reports, labels, and screens.

Examples:

```
? DATE()                    && Result: 09/01/87
? CDOW(DATE())              && Result: Tuesday
? CDOW(DATE() + 7)          && Result: Tuesday
? CDOW(CTOD("06/12/87"))    && Result: Friday
```

Library:

CLIPPER.LIB

See also:

DOW(), CMONTH(), MONTH(), DAY(), YEAR(), CTOD(), DTOC(), DTOS(), DATE()

CHR()

Syntax:

CHR(<expN>)

Purpose:

To return a character from the IBM Extended Character Set.

Argument:

<expN> is the IBM Extended Character Set code of the character to return and can be in the range of zero to 255.

Returns:

A character value.

CHR() returns the character corresponding to the IBM Extended Character Set code. Note that different commands treat characters in different ways. For example, @...SAY CHR(7) displays a graphic character to the screen where ? CHR(7) sounds the bell.

See the Appendix G for a complete list of available characters.

Usage:

CHR() is very versatile and serves a number of common tasks. The most typical use of CHR() is to send control codes to the printer. Another typical application is to use CHR() to ring the bell alerting the user of an error or the completion of an action. A third very common use is to send graphic characters to the screen or printer.

A more sophisticated application is to control the keyboard. Typically, you use CHR() in combination with KEYBOARD to stuff the keyboard with key codes. In some instances, you might find it useful to create key names by assigning the CHR() of the key code to a memory variable. Later, you can compare the result of a CHR(INKEY()) to the key name variable.

> **Note:** CHR(0) now has a length of one and is treated the same as any other character. This allows you to send it to any device or file including a database file.

Examples:

```
? CHR(72)                    && Result: H
? CHR(61)                    && Result: =
? REPLICATE(CHR(61), 10)     && Result: ==========
? CHR(ASC("A") + 32)         && Result: a
? CHR(7)                     && Result: bell sounds
```

```
*
? LEN(CHR(0))                    && Result: 1
? LEN("")                        && Result: 0
```

Library: CLIPPER.LIB

See also: KEYBOARD, ASC(), INKEY()

CMONTH()

Syntax:

CMONTH(<expD>)

Purpose:

To convert a date value to a string representing the name of the month.

Argument:

<expD> is the date value to convert.

Returns:

A character string.

CMONTH() returns the name of the month from a date value with the first letter in upper case and the rest of the string in lower case. The maximum return value length is nine characters for the month of September. A null date value returns a null string ("").

Usage:

CMONTH() is useful for creating formatted date strings that you can use in reports, labels, or screens.

Examples:

```
? CMONTH(DATE())                && Result: September
? CMONTH(DATE() + 45)           && Result: October
? SUBSTR(CMONTH(DATE()), 1, 3) +;
  STR(MONTH(DATE()))            && Result: Sep  1
```

Library:

CLIPPER.LIB

See also:

CDOW(), DOW(), MONTH(), DAY(), YEAR(), CTOD(), DTOC(), DTOS(), DATE()

COL()

Syntax: COL()

Purpose: To return the current screen column position of the cursor.

Returns: An integer numeric value.

Usage: COL() is used when you want to position the cursor to a column relative to the current column position. COL() is generally used in combination with ROW() and all variations of the @ command. In particular, you use it and ROW() to create screen position-independent procedures or functions where you pass the upper left row and column as parameters.

Example:

```
charvar = "This is"
@ 1, 10 SAY charvar
@ 1, COL() + 1 SAY "a string"
```

Results:

```
This is a string
```

Library: CLIPPER.LIB

See also: @...SAY...GET, @...BOX, @...CLEAR...TO, @...TO...[DOUBLE], ROW(), PCOL(), PROW()

CTOD()

Syntax:

CTOD(<expC>)

Purpose:

To convert a date string to a date value.

Argument:

<expC> is a character string consisting of numbers representing the month, day, and year separated by a delimiter character (any character other than a number). When you pass the date string, CTOD() evaluates the order of the month, day, and year substrings according to the DATE SETting. The default is AMERICAN ("mm/dd/yy").

Century: The twentieth century is the default if only two digits are specified for the year.

Empty date: To specify a null date, use SPACE(8), "", or " / / " as the function argument.

Returns:

A date value.

Usage:

CTOD() is useful whenever you want to use a character string as a date value. There are a number of common instances which include:

- Initializing a memory variable as a date.
- Specifying a literal date string as an argument of a RANGE clause of @...GET.
- Specifying a literal date string in order to perform date arithmetic.
- Comparing the result of a date expression to a literal date string.
- REPLACEing a date field with a literal date string.

Examples:

```
charvar  = "09/01/87"
? TYPE("charvar")                && Result: C
saledate = CTOD(charvar)
? TYPE("saledate")               && Result: D

SET DATE ANSI
? CTOD("12@12@12")            && Result: 12.12.12
? CTOD("12.12.12")            && Result: 12.12.12
? CTOD("12a12a12")            && Result: 12.12.12
? CTOD("12 12 12")            && Result: 12.12.12

? CTOD(SPACE(8))              && Result:   /  /
? CTOD("")                    && Result:   /  /
```

Library:

CLIPPER.LIB

See also:

SET DATE, CDOW(), DOW(), CMONTH(), MONTH(), DAY(), YEAR(), DTOC(), DTOS(), DATE()

DATE()

Syntax:

DATE()

Purpose:

To return the system date as a date value.

Returns:

A date value.

DATE() returns the system date in a format set by any combination of SET DATE and SET CENTURY. The default is AMERICAN and CENTURY OFF ("mm/dd/yy").

Usage:

DATE() provides a means of initializing memory variables to the current date, comparing other date values to the current date, and performing date arithmetic relative to the current date.

Examples:

```
? DATE()                    && Result: 09/01/87
? DATE() + 30               && Result: 10/01/87
? DATE() - 30               && Result: 08/02/87
datevar = DATE()
? CMONTH(datevar)           && Result: September

SET CENTURY ON
SET DATE ANSI
? DATE()                    && Result: 1987.09.01
```

Library:

CLIPPER.LIB

See also:

SET CENTURY, SET DATE, CDOW(), DOW(), CMONTH(), MONTH(), DAY(), YEAR(), CTOD(), DTOC(), DTOS()

CURDIR()

Syntax:

CURDIR([<expC>])

Purpose:

To determine the current DOS directory path of a specified drive.

Argument:

<expC> is the drive letter (A, B, ...). If omitted, the current DOS drive is assumed.

Returns:

A character string.

CURDIR() returns the DOS directory path of the drive specified by <expC>. If the return value is a null string ("), either an error has occurred, or the root directory is the current drive.

Examples:

```
? CURDIR("E:")    && Result: null string -- root
                     directory
? CURDIR("C")     && Result: FW\CLIPPER
? CURDIR("C:")    && Result: FW\CLIPPER
? CURDIR()        && Result: null string -- root
                     directory
? CURDIR("A")     && Result: null string -- drive not
                     ready
```

Library:

EXTEND.LIB

Source:

EXAMPLEA.ASM

See also:

SET DEFAULT, SET PATH

DAY()

Syntax:

DAY(<expD>)

Purpose:

To convert a date value to a number identifying the day of the month.

Argument:

<expD> is a date value to convert.

Returns:

An integer numeric value.

DAY() returns a number in the range of zero to 31 depending on the month of <expD>. If the month is February, leap years are accounted for and the number returned is either 28 or 29. In Clipper, if the date argument is February 29 and the year is not a leap year, the value returned is zero. If the date argument is empty, then DAY() returns zero.

Usage:

DAY() is useful when you want to use the day of a month during calculations.

Examples:

```
? DATE()                    && Result: 09/01/87
? DAY(DATE())               && Result: 1
? DAY(DATE()) + 1           && Result: 2
? DAY(CTOD(""))             && Result: 0
```

Library:

CLIPPER.LIB

See also:

CDOW(), DOW(), CMONTH(), MONTH(), YEAR(), CTOD(), DTOC(), DTOS(), DATE()

DBEDIT()

Syntax:

DBEDIT([<expN1> [,<expN2> [,<expN3> [,<expN4>]]]] [,<array1>] [,<expC>] [,<array2>] [,<array3>] [,<array4>] [,<array5>] [,<array6>] [,<array7>])

Purpose:

To display and edit records from one or more work areas using a browse-style table layout that executes within a defined window area.

Arguments:

<expN1...expN4> are the coordinates of the DBEDIT() window. Any, or all, of these arguments can be specified.

<array1> is an array of <expC> containing field names or expressions of any type. If this argument is not specified, DBEDIT() defaults to all fields in the current work area.

<expC> is a user-defined function that executes when a key exception is pressed or when there are no more keys to process in the keyboard buffer. Specify the function name without the parenthetical suffix or arguments. Note that the behavior of DBEDIT() is affected by the presence of this argument. Refer to the discussion below for more information.

<array2> is an array of <expC> to be used as picture strings for column formatting and is the same as TRANSFORM(). Specifying an <expC> instead of an array formats all columns with the same picture.

<array3> is an array of <expC> for column headings.

<array4> is an array of <expC> used to draw lines separating headings and the field display area. Specifying an <expC> instead of an array uses the same character for the heading line separator.

<array5> is an array of <expC> used to draw lines separating displayed columns. Specifying an <expC> instead of an array uses the same character for the column separator.

<array6> is an array of <expC> used to draw lines separating footings and the field display area. Specifying an <expC>

instead of an array uses the same character for the footing line separator.

<array7> is an array of <expC> to display as column footings. To force a column footing onto more than one line, embed a semicolon where you want the string to break. Specifying an <expC> instead of an array gives all footings the same value.

All arguments are optional. You must, however, pass a dummy argument for any argument you wish to skip.

Returns:

A logical value.

Usage:

DBEDIT() is an interface function that displays records in a table form. It is useful for full-screen editing of one or more data files. It formats the display according to the window coordinates and the fields array. All cursor movement keys are handled within DBEDIT. This includes PgUp, PgDn, Home, End, the four arrows, and all valid Ctrl key combinations that produce cursor movement. The following are the active keys with no function argument specified:

Table 6-6 DBEDIT() Active Keys

Key	Action
Uparrow	Up one row
Dnarrow	Down one row
Leftarrow	Column left
Rightarrow	Column right
Ctrl-Leftarrow	Pan left one column
Ctrl-Rightarrow	Pan right one column
Home	Leftmost current screen column
End	Rightmost current screen column
Ctrl-Home	Leftmost column
Ctrl-End	Rightmost column
PgUp	Previous screen
PgDn	Next screen
Ctrl-PgUp	First row of current column
Ctrl-PgDn	Last row of current column
Return	Terminate DBEDIT()
Esc	Terminate DBEDIT()

Note that DBEDIT() is not a wait state and so a SET KEY procedure cannot be invoked from within DBEDIT()'s native works unless you have your own wait state within the user function.

User Function: When the user function argument (<expC>) is specified, all keys indicated above are active with the exception of Esc and Return. When DBEDIT() calls the user function, it automatically passes two parameters: "status" and "fld_ptr." The status parameter indicates the current state of DBEDIT() depending on the last key executed before the user function was called. The following are the possible status values:

Table 6-7 DBEDIT() Status Messages

Status	Description
0	Idle, any cursor movement keystrokes have been handled and no keystrokes are pending
1	Attempt to cursor past beginning-of-file
2	Attempt to cursor past end-of-file
3	Database file is empty
4	Keystroke exception

Status messages 0, 1, 2 and 4 are used to process keys.

The other parameter, fld_ptr, is an index into the array of field names argument (<array1>). If <array1> is not specified, fld_ptr points to the current field in the database structure and can be accessed using FIELD().

When the user function has been called, you must return a value instructing DBEDIT() what action to perform next. The following table summarizes the possible request values and their consequences:

Table 6-7.1 Requests to DBEDIT() from User Function

Value	Description
0	Quit DBEDIT()
1	Continue DBEDIT()
2	Force reread/repaint and continue screen refresh; after refresh, go to idle
3	Append mode toggle

The user function is called in a number of different instances:

- A key exception occurs. This happens when DBEDIT() picks up a keystroke from the keyboard that is not a DBEDIT() executable key. Any pending keys remain in the keyboard buffer until picked up within the user function or DBEDIT() continues.

- DBEDIT() goes to idle. This happens when the keyboard is empty or when a request for a screen refresh has been executed (request = 2). In this instance, there is one call to the user function.

- Beginning or end-of-file are encountered. This is the same as idle. All executable keys are performed and then there is one call with the indicating status message.

Note that when DBEDIT() is first executed, all keys pending in the keyboard are executed and then DBEDIT() goes to idle with a user function call. If no keys are pending, the idle state is immediate.

The structure of the user function should be set up to handle all status messages received from DBEDIT(). Status messages characteristically indicate the current state of DBEDIT() and point to the type of action to be taken by the user function. The structure therefore should consist of a CASE structure that branches control to a subprocedure for each status message to process. Although you may be interested in less than all five of the status message values, your structure should at a minimum process idle states (status = 0) and key exceptions (status = 4). Ignored status messages can be processed by the

OTHERWISE clause. Tests for specific key presses should then take place within the called subprocedure. The following block of code demonstrates the basic structure of the user function:

```
FUNCTION UserFunc
PARAMETERS status, fld_ptr
PRIVATE request

key_stroke = LASTKEY()

DO CASE
CASE status = 0
    * Idle.
    request = ProcessIdle(key_stroke)

CASE status = 1
    * Beginning-of-file.
    request = ProcessBof(key_stroke)

CASE status = 2
    * End-of-file.
    request = ProcessEof(key_stroke)

CASE status = 3
    * Empty database file.
    request = ProcessEmpty()

CASE status = 4
    * Key exception.
    request = KeyExcept(key_stroke)

OTHERWISE
    request = 1

ENDCASE

RETURN request
```

There are two additional issues to deal with in the user function proper. First, to allow further processing of keys and the subsequent use of LASTKEY() within called subprocedures, save the value of the last key pressed to a memory variable before entering the main CASE structure. Then pass that key value throughout your subprocedure system as a parameter. Second, to allow execution of statements after the CASE structure, make your request to DBEDIT() by assigning the

request value to a memory variable and then RETURN the value as the last statement in the user function.

The two most important subprocedures in the user function architecture are the key exception handler and the idle state processor. In the key exception handler, you should account for user requests for the following set of actions:

- Exit from DBEDIT()
- Toggle delete status of the current record
- Field edit
- Entry into a menu system

In addition to these basic facilities, the key exception handler is where most of the activity of the user function system takes place. For example, the following demonstrates a typical key exception handling subprocedure:

```
FUNCTION KeyExcept
PARAMETER action_key

DO CASE
CASE action_key = 27
   * Esc...exit DBEDIT().
   RETURN 0

CASE action_key = 13
   * Return...field edit (see discussion below).
   RETURN FieldEdit()

CASE action_key = 7 .AND. (.NOT. EOF()) .AND.;
   LASTREC()<>0
   * Del...toggle delete status.
   IF DELETED()
      RECALL
   ELSE
      DELETE
   ENDIF
   RETURN 1

CASE action_key = -9
   * F10...enter menu system.
   RETURN MenuSys()

OTHERWISE
   * Any other key...slightly distasteful tone.
   TONE(100, 1)
```

```
    RETURN 1
ENDCASE
```

The other subprocedure of interest is the idle state processor. In the simplest case, you may just want to update a status area on the screen. In a more complex scenario, you can perform actions based on the last DBEDIT() key executed.

To edit a field value, add code like the following to the CASE structure in your key exception handling subprocedure.

```
CASE keystroke = 13      && Return key.

    * Save current key expression and value.
    index_exp = INDEXKEY(0)
    index_val = &index_exp.

    * Edit current field value.
    SET CURSOR ON
    field_name = field_list[fld_ptr]
    @ ROW(), COL() GET &field_name.
    READ
    SET CURSOR OFF

    * Refresh screen if key value has changed.
    RETURN IF(index_val  &index_exp., 2, 1)

CASE ...
```

The basic idea is to test for the key you are using for the field edit key and then GET the field pointed to by the fld_ptr parameter in the field array. This means that if you specify a field array as an argument to DBEDIT() you must use the same array name in your field edit routine. If you have not specified a field array, obtain the field name using FIELD() with fld_ptr as the argument.

A second issue is the appearance of the cursor. By default, the DBEDIT() cursor is OFF. You must therefore SET the CURSOR ON before executing the READ and SET it OFF after the READ.

A last issue is updating the screen if a key value has changed. To set this up, you must save both the controlling index key expression and its current value before editing the current field value. Then after the field edit is complete, send a request for a

screen refresh (a value of 2) if the new key value is the same as the old key value.

Nesting: You can call multiple copies of DBEDIT() within each copy of DBEDIT() allowing you multiple browse windows on the screen at one time.

Examples:

This second example demonstrates how to use DBEDIT() with a user function.

```
SELECT 2
USE Customer INDEX Customer
SELECT 1
USE Sales INDEX Sales
SET RELATION TO Cust_num INTO Customer

DECLARE field_list[4]
field_list[1] = "Branch"
field_list[2] = "Salesman"
field_list[3] = "Amount"
*
* Fields in another work area must include the
* alias.
fields[4] = "Customer->Customer"

DBEDIT(4, 0, 22, 79, field_list, "UserFunc")
```

For an advanced example that demonstrates a full-featured browse, see Browse() in Examplep.prg on the distribution disk. Note that Browse() is included in EXTEND.LIB so that to use it you need not compile and link it. The syntax is:

```
Browse([<window coordinates>])
```

Library:

EXTEND.LIB

See also:

@...SAY...GET, READ, ACHOICE(), MEMOEDIT()

DBFILTER()

Syntax:

DBFILTER()

Purpose:

To determine the expression of the active filter in the current work area.

Returns:

A character string.

DBFILTER() returns as a character string the filter condition defined in the current work area. If no FILTER has been SET, DBFILTER() returns a null string ("").

Example:

The following user-defined function, CreateQry(), uses DBFILTER() to create a memory file containing the current filter expression in the memory variable "qry_string." The memory file is named with the extension "qwy" to indicate that it contains a query.

```
FUNCTION CreateQry
PARAMETERS qry_name
*
qry_string = DBFILTER()
SAVE ALL LIKE qry_string TO &qry_name..qwy
RETURN .F.
```

You can later RESTORE a query file created by CreateQry() with the following user-defined function, SetFilter(). This function RESTOREs the query file and then SETs FILTER TO the condition stored in the variable "qry_string."

```
FUNCTION SetFilter
PARAMETERS qry_name
*
RESTORE FROM &qry_name..qwy ADDITIVE
SET FILTER TO &qry_string.
RETURN .F.
```

Library:

CLIPPER.LIB

See also:

DBRELATION(), DBRSELECT(), SET FILTER

DBRELATION()

Syntax:

DBRELATION(<expN>)

Purpose:

To determine the linking expression of a specified relation in the current work area.

Argument:

<expN> is the ordinal position in the list of relations defined.

Returns:

A character string.

DBRELATION() returns a character string containing the relation expression of the relation pointed to by <expN>. If there is no RELATION SET for <expN>, DBRELATION() returns a null string ("").

Usage:

DBRELATION() is used in combination with DBRSELECT() to query the linking expression and work area of an existing relation. Using these functions in addition to DBFILTER(), you can create a user-defined View system analgous to that of dBASE III PLUS and the Clipper utility, DBU.EXE. Essentially, you utilize these functions to create the View definition from the environment.

Example:

```
USE Customer INDEX Customer
SELECT 2
USE Invoices INDEX Invoices
SELECT 3
USE BackOrder INDEX BackOrder
SELECT 1
SET RELATION TO Cust_num INTO Customer;
    TO Cust_num INTO BackOrder
*
? DBRELATION(2)          && Result: Cust_num
```

Library:

CLIPPER.LIB

See also:

DBFILTER(), DBRSELECT(), SET RELATION

DBRSELECT()

Syntax:

DBRSELECT(<expN>)

Purpose:

To determine the target work area of a specified relation defined in the current work area.

Argument:

<expN> is the ordinal position in the list of relations defined.

Returns:

An integer numeric value.

DBRSELECT() returns the work area number of the relation pointed to by <expN>. If there is no RELATION SET for <expN>, DBRSELECT() returns zero.

Usage:

DBRSELECT() is used in combination with DBRELATION() to query the work area and linking expression of an existing relation. Using these functions along with DBFILTER(), you can create a user-defined View system analogous to that of dBASE III PLUS and the Clipper utility, DBU.EXE. Essentially, you utilize these functions to create the View definition from the environment.

Since the work area number is environment specific information, you may want the alias name if your application is independent of a specific work area. In this case, use the following expression to obtain the alias of the relation:

```
ALIAS(DBRSELECT(<expN>))
```

Example:

```
USE Customer INDEX Customer
SELECT 2
USE Invoices INDEX Invoices
SELECT 3
USE BackOrder INDEX BackOrder
SELECT 1
SET RELATION TO Cust_num INTO Customer;
   TO Cust_num INTO BackOrder
? DBRELATION(2)            && Result: Cust_num
? DBRSELECT(2)             && Result: 3
? ALIAS(DBRSELECT(2))      && Result: BACKORDER
```

Library:

CLIPPER.LIB

See also:

SET RELATION, DBFILTER(), DBRELATION()

DESCEND()

Syntax: DESCEND(<exp>)

Purpose: To create and SEEK descending order indexes.

Argument: <exp> is an expression of any data type.

Returns: DESCEND() returns the same data type as the <exp> in a complemented form.

Usage: DESCEND() is designed to be used in combination with INDEX and SEEK to allow for the creation of descending order indexes.

Examples: To use DESCEND() in an INDEX expression, use the following syntax:

```
INDEX ON DESCEND(Sales_date) TO date_dwn
```

To SEEK on the descending index, use the following syntax:

```
SEEK DESCEND(find_date)
```

Library: EXTEND.LIB

See also: INDEX, SEEK

DISKSPACE()

Syntax:

DISKSPACE([<expN>])

Purpose:

To determine the number of available bytes remaining on the specified disk drive.

Argument:

<expN> is the number of the drive to query where one is drive A, two is B, and three is C, etc. The default is the current drive if <expN> is omitted or zero.

Returns:

An integer numeric value.

DISKSPACE() returns the number of bytes of empty space on the specified disk drive. If DISKSPACE() is specified without an argument, the return value reflects the amount of space available on the default drive. It does not honor the SET DEFAULT drive.

Usage:

DISKSPACE() is useful when COPYing or SORTing to another drive and you want to determine if there is enough space available before initiating the operation. A variation of this concept is to use DISKSPACE() in combination with the RECSIZE() and RECCOUNT() functions to create a procedure to automatically backup database files.

Example:

This example is a user-defined function that demonstrates the use of DISKSPACE() to backup a database file to another drive.

```
FUNCTION BackUp
PARAMETERS outfile, drive
*
needed = INT((RECSIZE() * LASTREC()) + HEADER() + 1)
IF DISKSPACE() < needed
   RETURN .F.
ENDIF
COPY TO &drive.:&outfile.
RETURN .T.
```

Library:

EXTEND.LIB

Source: EXAMPLEC.C

See also: LUPDATE(), LASTREC()/RECCOUNT(), RECSIZE()

DOSERROR()

Syntax: DOSERROR()

Purpose: To determine the error number of the last DOS error.

Returns: An integer numeric value.

Usage: DOSERROR() is used in conjunction with the error function Open_error() to determine the exact cause of a file use error.

When your program experiences a DOS error, the error includes a number, described below.

Error Number	Description
1	Invalid function number
2	File not found
3	Path not found
4	Too many open files (no handles left)
5	Access denied
6	Invalid handle
7	Memory control blocks destroyed
8	Insufficient memory
9	Invalid memory block address
10	Invalid environment
11	Invalid format
12	Invalid access code
13	Invalid data
14	Reserved
15	Invalid drive was specified
16	Attempt to remove the current directory
17	Not same device
18	No more files
19	Attempt to write on write-protected diskette
20	Unknown unit
21	Drive not ready

Error number	Description
22	Unknown command
23	Data error (CRC)
24	Bad request structure length
25	Seek error
26	Unknown media type
27	Sector not found
28	Printer out of paper
29	Write fault
30	Read fault
31	General failure
32	Sharing violation
33	Lock violation
34	Invalid disk change
35	FCB unavailable
36	Sharing buffer overflow
37-49	Reserved
50	Network request not supported
51	Remote computer not listening
52	Duplicate name on network
53	Network name not found
54	Network busy
55	Network device no longer exists
56	Network BIOS command limit exceeded
57	Network adapter hardware error
58	Incorrect response from network
59	Unexpected network error
60	Incompatible remote adapter
61	Print queue full
62	Not enough space for print file
63	Print file deleted (not enough space)
64	Network name deleted
65	Access denied
66	Network device type incorrect
67	Network name not found
68	Network name limit exceeded
69	Network BIOS session limit exceeded
70	Temporarily paused
71	Network request not accepted
72	Print or disk redirection paused
73-79	Reserved
80	File already exists
81	Reserved

Error number	Description
82	Cannot make directory entry
83	Fail on INT 24H
84	Too many redirections
85	Duplicate redirection
86	Invalid password
87	Invalid parameter
88	Network device fault

Library: CLIPPER.LIB

See also: FERROR()

```
        READ
        *
        * Then, return a Rightarrow to advance cursor
        * on return to DBEDIT() and continue.  Note that
        * any cursor key can be returned by stuffing it
        * into the keyboard.
        KEYBOARD CHR(4)
        RETURN 1
    OTHERWISE
        *
        *  Don't quit.
        RETURN 1
    ENDCASE
    RETURN 1
```

Library: EXTEND.LIB

See also: @...SAY/GET, READ, ACHOICE()

DELETED()

Syntax: DELETED()

Purpose: To determine if the current record is marked for deletion.

Returns: A logical value.

DELETED() returns true (.T.) whenever the current record is marked for deletion; otherwise it returns false (.F.).

Usage: There are two areas where DELETED() may be used effectively. First is querying the deletion status as a part of record processing conditions. Second is displaying a record's deleted status as a part of screens and reports. To do this, format the return value of DELETED() using IF() as follows:

```
@ 1, 65 SAY IF(DELETED(), "Inactive", "Active")
```

Examples:

```
USE Sales
? RECNO()                    && Result: 1
? DELETED()                  && Result: .F.
DELETE
? DELETED()                  && Result: .T.
RECALL
? DELETED()                  && Result: .F.
```

Library: CLIPPER.LIB

See also: DELETE, RECALL, SET DELETED, PACK

DOW()

Syntax:

DOW(<expD>)

Purpose:

To convert a date value to a number identifying the day of the week.

Argument:

<expD> is a date value to convert.

Returns:

An integer numeric value.

DOW() returns a number between zero and seven. The first day of the week is one (Sunday) and the last is seven (Saturday). If <expD> is empty, DOW() returns zero.

Usage:

DOW() is useful when you want date calculations on a weekly basis. For example, you can use the DOW() to calculate last Monday's date with an expression like the following:

```
DATE() - DOW(DATE()) + 2
```

Examples:

```
? DATE()                    && Result: 09/01/87
? DOW(DATE())               && Result: 3
? CDOW(DATE())              && Result: Tuesday
? DOW(DATE() - 2)           && Result: 1
? CDOW(DATE() - 2)          && Result: Sunday
```

Library:

CLIPPER.LIB

See also:

CDOW(), CMONTH(), MONTH(), DAY(), YEAR(), CTOD(), DTOC(), DTOS(), DATE()

DTOC()

Syntax:

DTOC(<expD>)

Purpose:

To convert a date value to a character string.

Argument:

<expD> is the date value to convert.

Returns:

A character string.

DTOC() returns a character string representation of a date value based on the DATE and CENTURY SETtings. (See SET DATE for the supported formats.) The default format return value is in the form "mm/dd/yy." A null date returns a string of eight or ten spaces depending on whether CENTURY is OFF or ON.

Usage:

DTOC() is primarily useful for formatting purposes when you want to display the date in the SET DATE format and a character expression is required (in a LABEL FORM, for example). If you need a specialized date format, you can use TRANSFORM() or a custom expression.

If you are INDEXing a date as a part of a compound key, use DTOS() instead of DTOC().

Examples:

```
? DATE()                    && Result: 09/01/87
? DTOC(DATE())              && Result: 09/01/87
? "Today is " + DTOC(DATE())
```

Result:

```
Today is 09/01/87
```

Library:

CLIPPER.LIB

See also:

SET DATE, SET CENTURY, CDOW(), DOW(), CMONTH(), MONTH(), DAY(), YEAR(), CTOD(), DTOS(), DATE()

DTOS()

Syntax:

DTOS(<expD>)

Purpose:

To convert a date value to a character string suitable for INDEXing in a compound key.

Argument:

<expD> is the date value to convert.

Returns:

A character string.

DTOS() returns a string eight characters long in the format, "yyyymmdd." When <expD> evaluates to a null date, DTOS() returns a string of eight spaces.

Usage:

The intention of DTOS() is to facilitate the ease of creating index key expressions consisting of a date value and character expressions independent of DATE and CENTURY SETtings and preserving date order (year, month, and day).

Examples:

```
? DATE()                    && Result: 09/01/87
? DTOS(DATE())              && Result: 19870901
cdate = DTOS(DATE())
? TYPE("cdate")             && Result: C
? LEN(DTOS(CTOD("")))       && Result: 8
```

Library:

CLIPPER.LIB

See also:

INDEX, CDOW(), DOW(), CMONTH(), MONTH(), DAY(), YEAR(), CTOD(), DTOC(), DATE()

EMPTY()

Syntax: EMPTY(<exp>)

Purpose: To determine if the result of an expression is empty.

Argument: <exp> is an expression of any data type.

Returns: A logical value.

EMPTY() returns true (.T.), depending on the data type of the argument and according to the following criteria:

Table 6-8 List of Empty Values

Data Type	Contents
Character	Null or All spaces
Numeric	0
Date	Null
Logical	.F.

Examples:

```
? EMPTY(SPACE(5))          && Result: .T.
? EMPTY("")                && Result: .T.
? EMPTY(0)                 && Result: .T.
? EMPTY(CTOD(""))          && Result: .T.
? EMPTY(.F.)               && Result: .T.
*
USE Sales
APPEND BLANK
? EMPTY(Branch)            && Result: .T.
```

Library: CLIPPER.LIB

ERRORLEVEL()

Syntax:

ERRORLEVEL([<expN>])

Purpose:

To return the current DOS error level setting and optionally set the DOS error level to a new value.

Argument:

<expN> is the new DOS error level setting. This can be a value between zero and 255.

Returns:

An integer numeric value.

ERRORLEVEL() returns the current DOS error level setting.

Usage:

ERRORLEVEL() is primarily designed to use with SWITCH.EXE in order to chain execution of application programs. As an example, assume you have a main application program and, depending on some condition, it needs to execute another .EXE file. You can do this by setting the error level just before exiting to DOS as follows:

```
DO CASE
IF choice = "ONE"
   err_lev = 1
ELSEIF choice = "TWO"
   err_lev = 2
ELSEIF choice = "THREE"
   err_lev = 3
ENDIF
ERRORLEVEL(err_lev)        && Set error level.
QUIT
```

You then execute SWITCH from the DOS prompt specifying the list of executable application programs corresponding to error levels specified in the shell program. For example:

```
C>SWITCH ONE TWO THREE
```

Example:

```
error_level = ERRORLEVEL()     && Get current error
                               && level.
ERRORLEVEL(1)                  && Set new error level.
```

Library:

CLIPPER.LIB

EOF()

Syntax:

EOF()

Purpose:

To determine if an attempt has been made to move the record pointer past the end of the current database file.

Returns:

EOF() returns true (.T.) when you attempt to move the record pointer beyond the last logical record. When EOF() becomes true (.T.), the record pointer is positioned to LASTREC() + 1. This is the case whether or not an active FILTER is SET or DELETED is ON. Any further attempt to move the record pointer past LASTREC() + 1 returns the same result without error.

In addition, if the current database file contains no records, EOF() returns true (.T.).

Usage:

EOF() is used as a boundary condition test in any application where you are moving the record pointer downward through a database file.

Examples:

```
USE Sales
GO BOTTOM
? EOF()                && Result: .F.
SKIP
? EOF()                && Result: .T.
```

Library:

CLIPPER.LIB

See also:

FIND, SEEK, LOCATE, SKIP, BOF(), RECNO(), LASTREC(), FOUND()

EXP()

Syntax: EXP(<expN>)

Purpose: To calculate e^x where e is the base of natural logarithms and x is the numeric argument.

Argument: <expN> is the natural logarithm for which a numeric value is to be calculated.

Returns: A numeric value.

EXP() returns a value that obeys both the DECIMALS and FIXED SETtings.

Usage: EXP() is useful in equations requiring the use of the natural log function e^x.

EXP() is the inverse of LOG().

Examples:

```
? EXP(1)                    && Result: 2.72
SET DECIMALS TO 10
? EXP(1)                    && Result: 2.7182818285
? LOG(EXP(1))               && Result: 1.0000000000
```

Library: CLIPPER.LIB

See also: SET DECIMALS, SET FIXED, LOG()

FCLOSE()

Syntax: FCLOSE(<expN>)

Purpose: To close an open file, writing associated DOS buffers to disk.

Argument: <expN> is the file handle obtained previously from FOPEN() or FCREATE().

Returns: A logical value.

FCLOSE() returns false (.F.) if an error has occurred while writing, otherwise it returns true (.T.).

Example:

```
handle = FCREATE("Testfile", 0)
FCLOSE(handle)
```

Library: EXTEND.LIB

See also: FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

> **Warning:** These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FCOUNT()

Syntax:

FCOUNT()

Purpose:

To return the number of fields in the current database file structure.

Returns:

An integer value.

FCOUNT() returns the number of fields of the database file open in the current work area. If there is no database file open, FCOUNT() returns zero.

Usage:

FCOUNT() is useful in applications where you have created data-independent programs that can operate on any database file. These include generalized import/export and reporting programs. Typically, you can use FCOUNT() to establish the upper limit of FOR/NEXT or DO WHILE loops that process a single record at a time. For example, to display the names of all fields in the current work area:

```
FOR i = 1 TO FCOUNT()
   ? FIELD(i)
NEXT
```

Example:

```
USE Sales
? FCOUNT()                          && Result: 5
COPY STRUCTURE EXTENDED TO Temp
USE Temp
? LASTREC()                         && Result: 5
```

Library:

CLIPPER.LIB

See also:

FIELD()/FIELDNAME(), TYPE()

FCREATE()

Syntax:

FCREATE(<expC> [,<expN>])

Purpose:

To create a new file or truncate an existing file to zero length.

Arguments:

<expC> is the name of the file to create.

<expN> is the DOS file attribute. If omitted, the default is zero.

Table 6-9 DOS File Attributes

Value	Attribute	Description
0	Normal	Read/write.
1	Read only	Attempting to open for output returns an error.
2	Hidden	Excluded from normal directory searches.
4	System	Excluded from normal directory searches.

Returns:

A numeric value.

FCREATE() returns the DOS file handle number of the new file in the range of zero to 65,535. If an error occurs, it returns -1.

Usage:

When FCREATE() successfully creates a new file, it is left open with a DOS open mode of 2 (mode 2 is compatibility sharing mode, read/write access mode).

Since a file handle is required in order to identify an open file to other file functions, always assign the return value from FCREATE() to a memory variable for later use.

Accessing files in other directories: FCREATE() does not obey either the DEFAULT or PATH SETtings. Instead, it writes to the current directory unless a path is explicitly stated.

Example:

```
handle = FCREATE("Testfile", 0)
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

Warning: These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FERROR()

Syntax:

FERROR()

Purpose:

To test for a DOS error after a file function terminates.

Returns:

An integer numeric value.

FERROR() returns the DOS error from the last file operation (see Appendix F for a complete list of DOS errors). If there is no error, FERROR() returns zero.

Example:

```
shandle = FCREATE("Temp.txt")
IF FERROR() <> 0
   ? "Cannot create file, DOS error ", FERROR()
ENDIF
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FCREATE(), FOPEN(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

> **Warning:** These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FIELD()/FIELDNAME()

Syntax:

FIELD(<expN>)/FIELDNAME(<expN>)

Purpose:

To return the name of a specified field in the current database file.

Argument:

<expN> is the numeric position of a field in the database file structure.

Returns:

A character string.

If <expN> is not within the range of the fields contained in the current database file, FIELD() returns a null string ("").

Fieldnames are returned all upper case.

Usage:

FIELD() permits a database file structure to be handled as if it is an array, each field number a subscript pointer to the fieldname. This array of fields is useful for building data-independent programs for import/export and reporting.

If you need other database file structure information, use TYPE() and LEN(). If you need the number of decimal places a numeric field has defined, use the following expression:

```
LEN(SUBSTR(STR(<field>), RAT(".", STR(<field>)) + 1))
```

> **Note:** AFIELDS() provides similar but expanded capability for manipulating field attribute information.

Example:

```
USE Sales
? FIELD(1)                  && Result: BRANCH
? FCOUNT()                  && Result: 5
? LEN(FIELD(0))             && Result: 0
? LEN(FIELD(40))            && Result: 0
```

Library:

CLIPPER.LIB

See also:

COPY STRUCTURE EXTENDED, AFIELDS(), FCOUNT(), LASTREC(), TYPE()

FILE()

Syntax:

FILE(<expC>)

Purpose:

To determine if a file exists in the Clipper path.

Argument:

<expC> is the file to locate and must include an extension.

If the file is not in the default drive and directory, the Clipper path is searched. Be aware that the DOS path is not searched. If you do not have a Clipper PATH SET, you must explicitly specify the path, including the drive, as a part of the filename in order to access it.

Returns:

A logical value.

If the file exists, FILE() returns true (.T.).

Usage:

FILE() is useful in identifying a duplicate filename before information is stored to a file or for determining whether a needed file is available.

Examples:

```
? FILE("Sales.dbf")                    && Result: .F.
? FILE("\APPS\DBF\Sales.dbf")          && Result: .T.
SET PATH TO \APPS\DBF
? FILE("Sales.dbf")                    && Result: .T.
SET PATH TO
SET DEFAULT TO \APPS\DBF\
? FILE("Sales.dbf")                    && Result: .T.
```

Library:

CLIPPER.LIB

See also:

SET DEFAULT, SET PATH

FLOCK()

Syntax: FLOCK()

Purpose: To lock the shared database file in USE in the current work area.

Returns: A logical value.

If an attempt to lock a database file in USE in the currently selected work area succeeds, FLOCK() returns true (.T.). Otherwise it returns false (.F.).

Usage: The file lock remains until you UNLOCK, CLOSE the DATABASE, or RLOCK().

Note that unlike dBASE III PLUS, Clipper does not automatically lock all work areas in the relation chain when you lock the current work area. Likewise, an UNLOCK has no effect on related work areas.

Refer to Chapter 10 for more information on locking files and records.

Example:

```
IF FLOCK()
   DELETE ALL
ENDIF
```

Library: CLIPPER.LIB

See also: USE...EXCLUSIVE, SET EXCLUSIVE, UNLOCK, RLOCK()

FOPEN()

Syntax:

FOPEN(<expC> [,<expN>])

Purpose:

To open a file.

Arguments:

<expC> is the name of the file to open including the path if there is one.

<expN> is the requested DOS open mode indicating how the open file can be accessed. Access falls into three categories:

Table 6-10 DOS File Open Modes

Open Mode	Operation
0	Read only
1	Write only
2	Read/write

The default open mode is zero.

Returns:

A numeric value.

FOPEN() returns the file handle of the opened file in the range of zero to 65,535. If an error occurs, it returns -1.

Usage:

Since a file handle is required in order to identify an open file to other file functions, always assign the return value from FOPEN() to a memory variable for later use.

Accessing files in other directories: FOPEN() does not obey either the DEFAULT or PATH SETtings. Instead, it only searches the current directory unless a path is explicitly stated.

Example:

```
handle = FOPEN("Temp.txt")
IF FERROR() <> 0
   ? "Cannot open file, DOS error ", FERROR()
ENDIF
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FCREATE(), FERROR(), FREAD(), FREADSTR(), FSEEK(), FWRITE()

Warning: These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FOUND()

Syntax:

FOUND()

Purpose:

To determine if the previous search operation using FIND, LOCATE/CONTINUE, or SEEK succeeded.

Returns:

A logical value.

FOUND() returns true (.T.) if the last search command was successful.

Each work area has a FOUND() flag. This means that if one work area has a RELATION SET to a child work area, querying FOUND() in the child work area returns true (.T.) if there is a key value matching the current linking value from the parent.

Note that all commands other than search commands update FOUND() to false (.F.). This means that SKIP cannot be used with FOUND().

Usage:

FOUND() is useful for determining whether a search of a database file is successful before performing the next step in a program.

Examples:

```
USE Sales INDEX Sales
? INDEXKEY(0)                        && Result: SALESMAN
SEEK "100"
? FOUND()                            && Result: .T.
SEEK "1000"
? FOUND()                            && Result: .F.
* SEEK "100"
? FOUND()                            && Result: .T.
SKIP
? FOUND()                            && Result: .F.
* LOCATE FOR Branch = "100"
? FOUND()                            && Result: .T.
LOCATE FOR Branch = "1000"
? FOUND()                            && Result: .F.
```

Library:

CLIPPER.LIB

See also:

SEEK, FIND, LOCATE, CONTINUE, SET RELATION, SET SOFTSEEK, EOF()

FREAD()

Syntax:

FREAD(<expN1>, @<memvarC>, <expN2>)

Purpose:

To read characters from a file into a character memory variable.

Arguments:

<expN1> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<memvarC> is the name of an existing character memory variable passed by reference (preface it with the @ symbol) to use as a buffer. The length of this memory variable must be at least the same as <expN2>.

<expN2> is the number of bytes to read into the buffer starting at the current DOS pointer location. The value returned by a successful FREAD() should be equal to <expN2>.

Returns:

An integer numeric value.

FREAD() returns the number of bytes successfully read. A return value of zero indicates end-of-file or an error.

Usage:

FREAD() reads the file starting at the current file pointer position. Note that FREAD() reads all characters including control, null, and high-order (above CHR(128)).

To reposition the file pointer without reading, use FSEEK().

Example:

```
block  = 128
buffer = SPACE(512)
handle = FOPEN("Temp.txt")
*
IF FERROR() <> 0
   bytes = FREAD(handle, @buffer, block)
   IF bytes <> block
      ? "Error reading Temp.txt"
   ENDIF
ENDIF
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREADSTR(), FSEEK(), FWRITE()

Warning: These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FREADSTR()

Syntax:

FREADSTR(<expN1>, <expN2>)

Purpose:

To read characters from a file.

Arguments:

<expN1> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<expN2> is the number of bytes to read beginning at the current DOS file pointer position. This can be a positive or negative number depending on the direction (forward or backward) you want to read from the current pointer position.

Returns:

A character string.

FREADSTR() returns a string up to 65,535 (64K) bytes. A null return value ("") indicates an error or end-of-file.

Usage:

FREADSTR() reads from the current DOS file pointer position, the number of characters specified by <expN2> or until a null character (ASCII 0) is encountered. Like FREAD(), all characters are read including control characters.

Example:

The following example displays the ASCII value of the first 16 bytes of the text file, New.txt:

```
handle = FOPEN("New.txt")
IF FERROR() <> 0
   ? "File open error."
   RETURN
ELSE
   buffer = FREADSTR(handle, 16)
   ? "Length: ", LEN(buffer)
   ?
   FOR i = 1 TO LEN(buffer)
      ?? TRANSFORM(ASC(SUBSTR(buffer, i, 1)),; "99")
   NEXT
   FCLOSE(handle)
ENDIF
RETURN
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FSEEK(), FWRITE()

Warning: These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FSEEK()

Syntax:

FSEEK(<expN1>, <expN2> [,<expN3>])

Purpose:

To set the file pointer to a new position in a file.

Arguments:

<expN1> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<expN2> is the number of bytes to move the file pointer from the position as defined by <expN3>. This can be a positive or negative number depending on the direction to move the pointer.

<expN3> defines the method of moving and is indicated by a value from the following list:

Table 6-11 Methods of Moving the DOS File Pointer

Method	Description
0	Beginning-of-file
1	Current pointer position
2	End-of-file

The default method is zero.

Returns:

A numeric value.

FSEEK() returns the new position of the file pointer relative to the beginning-of-file.

Example:

```
handle = FOPEN("Temp.txt")
*
* Get length of the file.
length = FSEEK(handle, 0, 2)
*
* Reset file position.
FSEEK(handle, 0)
```

Library:

EXTEND.LIB

See also:

FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FWRITE()

Warning: These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

FWRITE()

Syntax:

FWRITE(<expN1>, <memvarC> [,<expN2>])

Purpose:

To write a buffer variable to a specified file.

Arguments:

<expN1> is the file handle obtained previously from FOPEN(), FCREATE(), or predefined by DOS.

<memvarC> is a pre-existing character memory variable to use as an output buffer.

<expN2> indicates the number of bytes to write from the buffer variable to the file beginning with the current file pointer position. If this argument is omitted, the entire contents of the buffer variable are written.

Returns:

A numeric value.

FWRITE() returns the number of bytes written. If the return value is zero, the disk is full or an error has occurred. Check FERROR() for the precise DOS error designation. Note that the value returned by a successful FWRITE() should be equal to <expN2>.

Example:

```
buffer = SPACE(512)
infile = FOPEN("Temp.txt")
*
outfile = FCREATE("Newfile.txt")
IF FERROR() <> 0
   ? "Cannot create file, DOS error ", FERROR()
   RETURN
ENDIF
*
FREAD(infile, @buffer, 512)
IF FERROR() <> 0
   ? "Cannot read file, DOS error ", FERROR()
   RETURN
ENDIF
*
FWRITE(outfile, buffer, 512)
IF FERROR() <> 0
   ? Cannot write file, DOS error ", FERROR()
ENDIF
```

GETE()

Syntax:

GETE(<expC>)

Purpose:

To retrieve the contents of a DOS environmental variable.

Argument:

<expC> is the name of the DOS environmental variable completely upper case. Note that when an environmental variable is SET, it is converted to upper case.

Returns:

A character string.

GETE() returns the contents of the specified DOS environmental variable. If the <expC> cannot be found, GETE() returns a null string ("").

> **Note:** If you are certain that an environmental variable exists and yet GETE() always returns a null string ("""), be sure there are no spaces between the environmental variable name and the first character of the string assigned to it.

Usage:

GETE() is useful for passing configuration information from the DOS environment into an application program. Typically, this could be configuration information that includes pointers to the location of files (database, index, label, or reports). This is particularly useful for network environments.

For example, when you set up a system define environmental variables that contain the location of various file types along with the CLIPPER environmental variable. The following DOS commands demonstrate:

```
C>SET LOC_DBF= <database file path>
C>SET LOC_NTX= <index file path>
C>SET LOC_RPT=<report file path>
```

In the configuration section of your application program, assign the contents of the environmental variables to memory variables. Then when you access a file, preface the reference with the path variable:

```
loc_dbf = GETE("LOC_DBF")
USE (loc_dbf + "<alias>")
```

Example:

```
path_set = GETE("PATH")
SET PATH TO &path_set.
```

Library:

EXTEND.LIB

Library: EXTEND.LIB

See also: FCLOSE(), FCREATE(), FERROR(), FOPEN(), FREAD(), FREADSTR(), FSEEK()

> **Warning:** These functions allow low level access to DOS files and devices. They should be used with extreme care and require a thorough knowledge of the operating system.

HARDCR()

Syntax: HARDCR(<expC>)

Purpose: To replace all soft carriage returns, CHR(141), with hard carriage returns, CHR(13) in order to display memo fields containing soft carriage returns.

Argument: <expC> is the character string or memo field to convert.

Returns: A character string.

HARDCR() returns a string up to 65,535 (64K) characters in length.

Example: To display a memo field formatted with the automatic word-wrapping of MEMOEDIT().

```
? HARDCR(Notes)
```

Library: EXTEND.LIB

See also: ?, @...SAY, REPORT FORM, MEMOEDIT(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT()

HEADER()

Syntax:

HEADER()

Purpose:

To determine the length of the header area of the current database file.

Returns:

An integer numeric value.

HEADER() returns the number of bytes in the header of the current database file.

Usage:

HEADER() can be used with RECCOUNT()/LASTREC(), RECSIZE() and DISKSPACE() to create procedures for backing up files.

Example:

```
USE Sales
? HEADER()                          && Result: 258

* Number of bytes in the current database file.
? (RECSIZE() * LASTREC()) +;
   HEADER() + 1                     && Result: 10339
```

Library:

EXTEND.LIB

See also:

DISKSPACE(), LASTREC()/RECCOUNT(), RECSIZE()

I2BIN()

Syntax:

I2BIN(<expN>)

Purpose:

To convert an integer numeric value to a character string formatted as an unsigned integer.

Argument:

<expN> is the integer number to convert. Note that decimal digits are truncated.

Returns:

A character string.

I2BIN() returns a two-byte character string as a 16-bit unsigned integer.

Usage:

I2BIN() is used in combination with FWRITE() to convert a Clipper numeric data type to a two-byte character string formatted as an unsigned integer.

Example:

This example opens a database file using low-level file functions and writes a new date of last update to bytes 1-3.

```
handle = FOPEN("Sales.dbf", 2)
*
* Convert date of last update to int.
year  = I2BIN(88)
month = I2BIN(12)
day   = I2BIN(15)
*
* Point to the date of last update.
FSEEK(handle, 1, 0)
*
* Write the new update date.
FWRITE(handle, year, 1)
FWRITE(handle, month, 1)
FWRITE(handle, day, 1)
*
FCLOSE(handle)
```

Library:

EXTEND.LIB

Source:

EXAMPLEA.ASM

See also:

BIN2I(), BIN2W(), BIN2L(), CHR(), L2BIN(), FOPEN(), FWRITE()

IF()/IIF()

Syntax:

IF(<expL>, <exp1>, <exp2>)/IIF(<expL>, <exp1>, <exp2>)

Purpose:

To return the result of one of two specified expressions, depending upon the logical state of the given condition.

Arguments:

<expL> is a logical expression to be evaluated.

<exp1> is the value to return if <expL> is true (.T.).

<exp2> is the value to return if <expL> is false (.F.).

Note that unlike other dialects, <exp1> and <exp2> can evaluate to different data types. Note also that only the path pointed to by the result of <expL> is evaluated. This means that an undefined or erroneous argument for the non-executed path does not generate a runtime error.

Returns:

A value of any data type.

IF() returns the evaluation of the argument pointed to by the result of the logical expression argument. Since the two return arguments can be different data types, the value returned is the data type of the evaluated return argument.

Usage:

IF() is one of the most powerful and versatile functions in Clipper. It provides a mechanism to evaluate a condition within an expression. With this ability you can convert a logical data type expression to another data type. For example, the following converts a logical to a numeric:

```
IF(<condition>, 0, 1)
```

This leads to a number of applications. You can, for example, format a logical field:

```
IF(Paid, SPACE(0), "Go get'em")
```

If you are printing forms, you may want to print an indicating symbol in different columns depending on the value of a logical field. For example:

```
@ <row>, IF(In_hosp, 10, 12) SAY "X"
```

INDEXing is another area where IF() is useful. You may want to create a key based on a logical field or add a key depending on a condition. This latter might be the case if the key field is empty. In this case, you want the contents of another field to be the key value. For example:

```
INDEX ON IF(EMPTY(Name), Company, Name);
   TO <ntx file>
```

You can also use IF() to force the LABEL FORM to print blank lines. For example:

```
IF(EMPTY(<expC>), CHR(255), <expC>)
```

Examples:

```
a = 100
? IF(a > 50, "greater", "less")      && Result: greater
a = 10
? IF(a > 50, "greater", "less")      && Result: less
? IF(a > 50, .T., 0)                 && Result: 0
? IF(.F., "Bad" + 12, "Good")        && Result: Good
```

Library: CLIPPER.LIB

See also: IF, DO CASE

INDEXEXT()

Syntax:

INDEXEXT()

Purpose:

To determine whether the current application was linked using NDX.OBJ for dBASE III PLUS compatible indexes.

Returns:

A character string.

INDEXEXT() returns "NDX" if the index file type you use is dBASE III PLUS compatible or "NTX" if it is Clipper compatible.

Example:

```
* Adds "NTX" or "NDX" to "Name.".
IF .NOT. FILE("Name." + INDEXEXT())
   INDEX ON Field1 TO Name
ENDIF
```

Library:

CLIPPER.LIB

See also:

INDEXKEY(), INDEXORD()

INDEXKEY()

Syntax:

INDEXKEY(<expN>)

Purpose:

To determine the key expression of a specified index.

Argument:

<expN> is the ordinal position of the index in the list of index files opened by the last USE...INDEX or SET INDEX TO command for the current work area. A zero value points to the controlling index, no matter what its actual position in the list.

Returns:

A character string.

INDEXKEY() returns the key expression of the specified index. If there is no index for the position the function argument points to, INDEXKEY() returns a null string ("").

Examples:

```
USE Customers INDEX Name, Serial
SET ORDER TO 2
? INDEXKEY(1)           && Result: Name index exp
? INDEXKEY(2)           && Result: Serial index exp
? INDEXKEY(0)           && Result: Serial index exp
```

Library:

CLIPPER.LIB

See also:

USE, SET INDEX, SET ORDER, INDEXEXT(), INDEXORD()

INDEXORD()

Syntax:

INDEXORD()

Purpose:

To determine the position of the controlling index in the list of index files opened by the last USE...INDEX or SET INDEX TO in the current work area.

Returns:

An integer numeric value.

INDEXORD() returns the position of the controlling index in the list of open index files. A value of zero indicates there is no controlling index and you are accessing the current database file in natural order.

Usage:

INDEXORD() is useful to record the last controlling index allowing you to restore that order sometime later.

Example:

```
USE Customers INDEX Name, Serial
last_ord = INDEXORD()          && Result: 1
SET ORDER TO 2
? INDEXORD()                   && Result: 2
SET ORDER TO last_ord
? INDEXORD()                   && Result: 1
```

Library:

CLIPPER.LIB

See also:

USE, SET INDEX, SET ORDER, INDEXEXT(), INDEXKEY()

INKEY()

Syntax:

INKEY([<expN>])

Purpose:

To read a character from the keyboard.

Argument:

<expN> specifies the number of seconds INKEY() waits for a key press. Specifying zero halts the program until a key is pressed. Note that the time INKEY() waits is based on the operating system clock and therefore not related to the microprocessor speed.

Returns:

An integer numeric value.

INKEY() returns a number from -39 to 386, identifying the ASCII code of the key pressed (the same value as returned by LASTKEY()). If the keyboard buffer is empty, INKEY() returns zero.

INKEY() returns values for all function, Alt-function, Ctrl-function, Alt-letter, and Ctrl-letter key combinations. See Appendix G for the complete list of key values.

Usage:

INKEY() is useful for polling the keyboard or pausing program execution. As an instance, you can use INKEY() to terminate commands with a record scope such as LIST, LABEL FORM, and REPORT FORM by including it in a WHILE condition as follows:

```
REPORT FORM Report WHILE INKEY() <> 27
```

To make your key operations easier to maintain, create a series of keyname memory variables using CHR(). Later in your program, you can compare the result of INKEY() to the keyname variable using the expression, CHR(INKEY()).

Example:

The following displays the character and the code after a key is pressed.

```
i = 0
DO WHILE LASTKEY() <> 27
   ? "Press any key: "
   i = INKEY(0)
   ?? "Character:", CHR(i), "ASCII code:",;
      LTRIM(STR(i))
ENDDO
RETURN
```

Library:

CLIPPER.LIB

See also:

SET KEY, CHR(), LASTKEY()

INT()

Syntax: INT(<expN>)

Purpose: To convert a numeric expression to an integer by truncating all digits to the right of the decimal point.

Argument: <expN> is a numeric expression to convert to an integer.

Returns: An integer numeric value.

INT() does not round the argument expression.

Usage: INT() is useful in operations where the decimal value portion of a number is not needed.

Examples:

```
? INT(100.00)          && Result: 100
? INT(.5)              && Result: 0
? INT(-100.00)         && Result: -100
```

Library: CLIPPER.LIB

See also: ROUND()

ISALPHA()

Syntax:

ISALPHA(<expC>)

Purpose:

To determine if the specified character string begins with an alphabetic character.

Argument:

<expC> is the character string to examine.

Returns:

A logical value.

ISALPHA() returns true (.T.) if the first character in <expC> is alphabetic. An alphabetic character consists of any upper case or lower case letter from A to Z. ISALPHA() returns a logical false (.F.) if a string begins with a number or any other character.

Examples:

```
? ISALPHA("AbcDe")          && Result: .T.
? ISALPHA("aBcDE")          && Result: .T.
? ISALPHA("1BCde")          && Result: .F.
? ISALPHA(".FRED")          && Result: .F.
```

Library:

EXTEND.LIB

See also:

ISLOWER(), ISUPPER(), LOWER(), UPPER()

ISCOLOR()

Syntax:

ISCOLOR()

Purpose:

To determine if the computer running a Clipper-compiled program has a color graphics card installed.

Returns:

A logical value.

ISCOLOR() returns a true (.T.) if there is a color graphics card installed.

Usage:

ISCOLOR() allows you to make decisions about type of screen attributes to assign (color or monochrome). Note that some monochrome adapters with graphics capability return true (.T.).

Example:

The following installs color attributes variables at runtime:

```
IF ISCOLOR()
   c_box  = "BG+/B, W/N"
   c_says = "BG/B,  W/N"
   c_gets = "W/N,   N/W"
ELSE
   c_box  = "W+"
   c_says = "W/N, N+/W"
   c_gets = "W/N, N/W"
ENDIF
```

Library:

CLIPPER.LIB

See also:

SET COLOR

LASTKEY()

Syntax:

LASTKEY()

Purpose:

To determine the last key pressed in a wait state.

Returns:

An integer value.

LASTKEY() returns a number from -39 to 386, identifying the ASCII value of the last key entered in a wait state. Wait state commands and functions include ACCEPT, INPUT, MENU TO, READ, WAIT, and INKEY().

See Appendix G for the values returned.

Usage:

LASTKEY() is most useful for determining the key used to terminate a READ. Additionally, if you are executing a user-defined function with the VALID clause, you can query the key pressed to leave the current GET. Based on the return value, you can make various types of decisions.

If you need to know whether the user has changed the contents of any GETs, use UPDATED().

Example:

```
USE Customer
m_bal = Bal
@  7, 10 SAY TRIM(Last) + ", " + TRIM(First)
@  9, 10 SAY "Current Balance";
           GET m_bal
@ 11, 10 SAY "Press <Esc> to discard change."
READ

* If the key used to stop editing was not Esc
* REPLACE the database field.
IF LASTKEY() <> 27
   REPLACE Bal WITH m_bal
ENDIF
```

Library:

CLIPPER.LIB

See also:

KEYBOARD, INKEY(), CHR()

ISLOWER()

Syntax:

ISLOWER(<expC>)

Purpose:

To determine if the leftmost character in the specified character string is lower case.

Argument:

<expC> is the character string to examine.

Returns:

A logical value.

ISLOWER() returns true (.T.) if the first character of the character string is lower case. Any other character returns false (.F.).

Examples:

```
? ISLOWER("aBcDe")          && Result: .T.
? ISLOWER("AbcDe")          && Result: .F.
```

Library:

EXTEND.LIB

See also:

ISALPHA(), ISUPPER(), LOWER(), UPPER()

ISPRINTER()

Syntax: ISPRINTER()

Purpose: To determine whether LPT1 is ready.

Returns: A logical value.

ISPRINTER() returns true (.T.) if LPT1 is ready; it returns false (.F.) if it is not.

Example:

```
@ 22, 17 SAY "Press any key to print"
@ 23, 38 SAY "or Esc to abort..."
key = 0
DO WHILE key <> 27 .AND. (.NOT. ISPRINTER())
   key = INKEY(0)
ENDDO
@ 22, 00 CLEAR
```

Library: EXTEND.LIB

Source: EXAMPLEA.ASM

See also: SET DEVICE, SET PRINT

ISUPPER()

Syntax:

ISUPPER(<expC>)

Purpose:

To determine whether the leftmost character in a character string is upper case.

Argument:

<expC> is the character string to examine.

Returns:

A logical value.

ISUPPER() returns true (.T.) if the first character is upper case; otherwise, it returns false (.F.).

Examples:

```
? ISUPPER("AbCdE")          && Result: .T.
? ISUPPER("aBCdE")          && Result: .F.
```

Library:

EXTEND.LIB

See also:

ISALPHA(), ISLOWER(), LOWER(), UPPER()

L2BIN()

Syntax: L2BIN(<expN>)

Purpose: To convert an integer numeric value to a character string formatted as a 32-bit signed integer.

Argument: <expN> is the integer number to convert. Note that decimal digits are truncated.

Returns: A character string.

L2BIN() returns a four-byte character string formatted as a 32-bit signed integer.

Usage: L2BIN() is used in combination with FWRITE() to convert numeric values to a form that can be written to a file that contains data formatted in binary format.

Example:

```
? BIN2L(L2BIN(1200))        && Result: 1200
```

Library: EXTEND.LIB

Source: EXAMPLEA.ASM

See also: BIN2I(), BIN2W(), BIN2L(), CHR(), I2BIN(), FOPEN(), FWRITE()

LASTKEY()

Syntax: LASTKEY()

Purpose: To determine the last key fetched from the keyboard buffer.

Returns: An integer value.

LASTKEY() returns a number from -39 to 386 identifying the ASCII value of the last key fetched. There are two classes of commands and functions that fetch keys: wait states and interface functions. Wait state commands and functions include ACCEPT, INPUT, MENU TO, MEMOEDIT(), READ, and WAIT. Interface functions include ACHOICE(), DBEDIT(), and INKEY().

See Appendix G for the values returned.

Usage: LASTKEY() is useful in a number of instances:

- To determine the key used to terminate a READ.
- To determine the key used to exit the current GET when within a user-defined function executed from a VALID clause.
- To identify the exception key pressed in the user function of ACHOICE(), DBEDIT(), or MEMOEDIT().

If you need to know whether the user has changed the contents of any GETs, use UPDATED(). If you need to know the key pending in the keyboard buffer, use NEXTKEY().

Example:

```
USE Customer
M->bal = Bal
@ 10, 10 SAY "Current Balance"
   GET M->bal
READ

* If the key used to stop editing was not Esc
* REPLACE the database field.
IF LASTKEY() <> 27
   REPLACE Bal WITH M-bal
ENDIF
```

Library: CLIPPER.LIB

See also: KEYBOARD, CHR(), INKEY(), NEXTKEY()

LASTREC() / RECCOUNT()

Syntax:

LASTREC() / RECCOUNT()

Purpose:

To determine the number of physical records in the current database file.

Returns:

An integer numeric value.

LASTREC() returns the number of physical records in the current database file. This means that filtering commands such as SET FILTER or SET DELETED have no affect on the return value.

LASTREC() returns zero if there is no database file in USE in the current work area.

Examples:

```
USE Sales
? LASTREC()                          && Result: 84
? RECCOUNT()                         && Result: 84
*
SET FILTER TO Salesman = "1001"
COUNT TO num_recs
? num_recs                           && Result: 14
? RECCOUNT()                         && Result: 84
```

Library:

CLIPPER.LIB

See also:

FIELD()

LEN()

Syntax:

LEN(<expC>/<array>)

Purpose:

To return the length of a character expression or the number of elements in an array.

Arguments:

<expC> is the character string to determine the length of.

<array> is the array to count.

Returns:

An integer numeric value.

If the character expression evaluates to a null string, LEN() returns zero.

Examples:

```
? LEN("string of characters")        && Result: 20
? LEN(SPACE(10))                     && Result: 10
? LEN(SPACE(0))                      && Result: 0
? LEN(TRIM(SPACE(10)))               && Result: 0

DECLARE test_array[10]
? LEN(test_array)                    && Result: 10
```

Library:

CLIPPER.LIB

See also:

DECLARE, LTRIM(), TRIM()/RTRIM()

LEFT()

Syntax:

LEFT(<expC>, <expN>)

Purpose:

To extract a specified number of characters from the left of a character string.

Arguments:

<expC> is a character string from which to extract characters.

<expN> is the number of characters to extract.

Returns:

A character string.

LEFT() returns the leftmost <expN> characters of <expC>. If <expN> is negative or is zero, LEFT() returns a null string ("). If <expN> is larger than the length of the character string, LEFT() returns the entire string. The maximum size of <expC> is 65,535 (64K) bytes.

Example:

```
? LEFT("ABCDEF", 3)                && Result: ABC
```

Library:

EXTEND.LIB

See also:

AT(), LTRIM(), RAT(), RIGHT(), RTRIM(), STUFF(), SUBSTR(), TRIM()

LOG()

Syntax:

LOG(<expN>)

Purpose:

To calculate the natural logarithm of a numeric expression.

Argument:

<expN> is a number greater than zero to convert to its natural logarithm.

Returns:

A numeric value.

Specifying a number less than or equal to zero returns a numeric overflow (a row of asterisks).

Usage:

The natural logarithm has a base of e which is 2.7183. The LOG() function returns "x" in the following equation:

$e^x = y$

"y" is the specified numerical expression. Due to mathematical rounding, the values returned by LOG() and EXP() may not agree exactly.

LOG() is the inverse of the EXP() function.

Examples:

```
? LOG(10)                    && Result: 2.30
? LOG(10 * 2)                && Result: 3.00
? EXP(LOG(1))                && Result: 1.00
? LOG(2.71)                  && Result: 1.00
```

Library:

CLIPPER.LIB

See also:

SET DECIMALS, SET FIXED, EXP()

LOWER()

Syntax: LOWER(<expC>)

Purpose: To convert upper case characters contained in the specified character string to lower case.

Argument: <expC> is a character expression to be converted to lower case.

Returns: A character string.

LOWER() returns lower case of all alphabetic characters in the argument. All other characters are ignored.

Examples:

```
? LOWER("STRING")              && Result: string
? LOWER("1234 CHARS = ")       && Result: 1234 chars =
```

Library: CLIPPER.LIB

See also: UPPER()

LUPDATE()

Syntax:

LUPDATE()

Purpose:

To determine the date the database file in the current work area was last modified and CLOSEd.

Returns:

A date value.

LUPDATE() returns the date of change of the current database file. If there is no database file in USE in the current work area, LUPDATE() returns a blank date. Note that the last date of change is not reflected until the database file is CLOSEd.

Examples:

```
? DATE()                    && Result: 09/01/87
USE Sales
? LUPDATE()                 && Result: 08/31/87
*
APPEND BLANK
? LUPDATE()                 && Result: 08/31/87
CLOSE DATABASES
*
USE Sales
? LUPDATE()                 && Result: 09/01/87
```

Library:

EXTEND.LIB

See also:

FIELD()/FIELDNAME(), LASTREC()/RECCOUNT(), RECSIZE()

LTRIM()

Syntax:

LTRIM(<expC>)

Purpose:

To remove leading blanks from the specified character string.

Argument:

<expC> is the character string to return with leading spaces removed.

Returns:

A character string.

If the argument is a null string ("""), LTRIM() returns a null also.

Usage:

LTRIM() is useful for text formatting of character strings that have leading spaces. These can be, for example, numbers converted to character strings using STR().

Examples:

```
number = 18
? STR(number)                    && Result:          18
? LEN(STR(number))               && Result: 10
? LTRIM(STR(number))             && Result: 18
? LEN(LTRIM(STR(number))         && Result: 2
```

Library:

CLIPPER.LIB

See also:

TRIM(), STR(), SUBSTR()

MAX()

Syntax:

MAX(<exp1>, <exp2>)

Purpose:

To determine the larger of two numeric or date expressions.

Arguments:

<exp1> is the first numeric or date expression to be compared.

<exp2> is the second numeric or date expression to be compared.

Note that both arguments must be the same type.

Returns:

A numeric or date value.

MAX() always returns the larger of the two arguments. This result is not affected by an empty or null argument.

Usage:

MAX() is useful when you want to insure that the value of an expression is larger than a specified minimum.

Examples:

```
? MAX(1, 2)                    && Result: 2
? MAX(2, 1)                    && Result: 2
*
? DATE()                       && Result: 09/01/87
? MAX(DATE(), DATE() + 30)     && Result: 10/01/87
? MAX(DATE(), CTOD(""))        && Result: 09/01/87
```

Library:

CLIPPER.LIB

See also:

MIN()

MEMOEDIT()

Syntax:

MEMOEDIT([<expC1>] [,<expN1>, <expN2>, <expN3>, <expN4>] [,<expL1>] [,<expC2>] [,<expN5>] [,<expN6>] [,<expN7>] [,<expN8>] [,<expN9>] [,<expN10>])

Purpose:

To display or edit character strings and memo fields.

Arguments:

<expC1> is the character string or memo field to edit.

<expN1>, <expN2>, <expN3>, <expN4> define the edit window coordinates in the following order: top, left, bottom, and right. If omitted, the entire screen is used.

<expL1> determines whether a memo is edited or simply displayed. If you specify true (.T.), the memo is displayed and you enter the edit mode. If you specify false (.F.), you are placed in the browse mode where the memo is simply displayed. The default is true (.T.).

<expC2> is the name of a user function (a Clipper user-defined function) to execute whenever a key is pressed. Specify the function name without the parenthetical suffix or arguments. Refer to the discussion below for more information.

<expN5> determines the line length. If <expN5> is greater than the width of the window (<expN4> - <expN2> - 1), the window scrolls horizontally. The default is (<expN4> - <expN2> - 1).

<expN6> determines the tab size and enables real tabs. The default is four.

<expN7> is the initial memo line where the cursor is placed.

<expN8> is the initial memo column where the cursor is placed.

<expN9> is the initial row for placing the cursor relative to the window position. The default is zero.

<expN10> is the initial column for placing the cursor relative to the window position. The default is zero.

All arguments are optional. You must, however, pass a dummy argument for any argument you wish to skip.

Returns:

A character string.

MEMOEDIT() returns the modified string if it is terminated with Ctrl-W or the original string if it is terminated with Esc.

Usage:

MEMOEDIT() is a general purpose text editing function you can use in your applications for a variety of purposes. It supports a number of different modes and includes a user function to allow key reconfiguration and other activities germane to programming the current text editing task.

The following are the active keys within MEMOEDIT():

Table 6-12 MEMOEDIT() Navigation Keys

Key	Purpose
Uparrow or Ctrl-E	Move up one line
Dnarrow or Ctrl-X	Move down one line
Leftarrow or Ctrl-S	Move left one character
Rightarrow or Ctrl-D	Move right one character
Ctrl-Leftarrow or Ctrl-A	Move left one word
Ctrl-Rightarrow or Ctrl-F	Move right one word
Home	Beginning of current line
End	End of current line
Ctrl-Home	Beginning of current window
Ctrl-End	End of current window
PgUp	Previous edit window
PgDn	Next edit window
Ctrl-PgUp	Beginning of memo
Ctrl-PgDn	End of memo

Table 6-13 MEMOEDIT() Editing Keys

Key	Purpose
Ctrl-Y	Delete the current line
Ctrl-T	Delete word right
Ctrl-B	Reformat memo in the edit window

Table 6-14 MEMOEDIT() Escape Keys

Key	Purpose
Ctrl-W	Finish editing with save
Esc	Abort edit and return original

Browse/Update modes: MEMOEDIT() supports two display modes depending on the value of <expL1>. If <expL1> is true (.T.), MEMOEDIT() enters update (edit) mode; otherwise, MEMOEDIT() enters browse (display) mode. In the browse mode, all navigation keys are active and perform the same actions as update mode with one exception. In update mode, the scroll state is off (Uparrow and Dnarrow move the cursor up or down one line). In browse mode, the scroll state is on (Uparrow and Dnarrow scroll the contents of the MEMOEDIT() window up or down one line).

Note that browse mode in Autumn '86 did not allow cursor movement and terminated immediately. If you wish to retain this behavior, add a user function argument for the following function:

```
FUNCTION NoBrowse
PARAMETERS mode
IF mode = 3
   KEYBOARD CHR(27)
   RETURN 0
ELSE
   RETURN 0
ENDIF
```

You can also retain the behavior by using the syntax: MEMOEDIT(<expC1>, <expN1>, <expN2>, <expN3>, <expN4>, .F., .F.)

When MEMOEDIT() executes, it automatically calls the user function, as explained below, which in this case immediately terminates MEMOEDIT(). Follow the MEMOEDIT() call with INKEY(0) to pause the display.

User Function: When MEMOEDIT() calls the user function, it automatically passes three parameters: "status," "line," and

"col." The status message indicates the current state of MEMOEDIT() depending on the last key pressed or the last action taken prior to executing the user function. The following status values are possible:

Table 6-14.1 MEMOEDIT() Status Messages

Status	Description
0	Idle
1	Re-configurable or unknown keystroke (memo unaltered)
2	Re-configurable or unknown keystroke (memo altered)
3	Start-up

A status value of 3 indicates that the current mode is the start-up mode. When you specify a user function, MEMOEDIT() makes a call to it immediately after being invoked. At this point, you RETURN a request to configure MEMOEDIT()'s various toggle states: word-wrap, scroll, or insert. MEMOEDIT() calls the user function repeatedly, remaining in the start-up mode until you RETURN 0. The memo is then displayed and you enter the display mode set by <expL1>.

Status messages 0, 1, and 2 are used to process keys. The idle state (status = 0) is called once when there is no pending key to process. Within this state, you generally update line and column number displays. MEMOEDIT() calls the user function whenever a key exception occurs. Keys that instigate a key exception are all available control keys, function keys, and Alt keys. Since these keys are not processed by MEMOEDIT() when you have a user function, they can all be re-configured.

The other two parameters, line and col, indicate the current cursor position in the MEMOEDIT() window when the user function is called. The line parameter begins with position one and col begins with position zero.

When the status is either 1, 2, or 3, you can return a value instructing MEMOEDIT() what action to perform next. The following table summarizes the possible return values and their consequences:

Table 6-14.2 MEMOEDIT() User Function Requests

Value	Action
0	Perform default action
1 - 31	Perform requested action corresponding to key value (e.g., 22 = Ctrl-V = Ins = toggle insert mode)
32	Ignore the current key (disable)
33	Process the current key as data (insert control key)
34	Toggle word-wrap
35	Toggle scrolling
The following exceptions resolve key value collisions:	
100	Next word (2 = Ctrl-B = reform)
101	Bottom right of window (23 = Ctrl-W = save and exit)

Note that cursor keys, Return, Backspace, Tab, Del, and character keys cannot be disabled.

Word-wrapping: Word-wrapping is a state you toggle by RETURNing a 34 from the user function. The default is on. When word-wrap is on, MEMOEDIT() inserts a soft carriage return/line feed at the closest word break to the window border or line length which ever occurs first. When word-wrap is off, MEMOEDIT() scrolls text entry beyond the window definition until you reach the end-of-line. At this point, you must press Return (inserting a hard carriage return/line feed) to advance to the next line.

Note that soft carriage returns may interfere with the result of display commands such as ? and REPORT FORM or processing with another word processor. Use HARDCR() and MEMOTRAN() to replace these embedded characters as needed.

Paragraph reform: Pressing Ctrl-B or RETURNing a 2 from a user function reformats the memo until a hard carriage (end-of-paragraph) or the end-of-memo is encountered. This happens regardless of whether word-wrap is on or off.

Tab characters: When you specify the tab size argument (<expN6>), MEMOEDIT() inserts a hard tab character (09H) in the text when Tab is pressed. If the tab size argument has not been specified, MEMOEDIT() inserts space characters instead. The size of tabs is global for the entire memo and set with <expN6>. The default is four.

Note that MEMOEDIT() does not convert tab characters to spaces if real tabs are on.

Examples:

To display a memo field without editing:

```
SET CURSOR OFF
MEMOEDIT(Memo, 5, 10, 20, 69, .F.)
SET CURSOR ON
```

To edit the current memo field:

```
REPLACE Memo WITH;
MEMOEDIT(Memo, 5, 10, 20, 69, .T.)
```

To create a character string using the full screen:

```
note = MEMOEDIT()
```

This example demonstrates a user-defined function that edits a memo field in a box with a title:

```
FUNCTION EditMemo
PARAMETERS memo, title, tr, tc, br, bc
PRIVATE temp_scr
SAVE SCREEN TO temp_scr
@ tr - 1, tc - 2 CLEAR TO br + 1, bc + 2
@ tr - 1, tc - 2 TO br + 1, bc + 2
@ tr - 1, tc SAY "[" + title + "]"
var  = MEMOEDIT(memo, tr, tc, br, bc, .T.)
RESTORE SCREEN FROM temp_scr
RETURN (var)
```

For an advanced example that demonstrates a full-featured MEMOEDIT() using a user function, see Me.prg on the distribution disk. Note that you must compile and link it before using it.

Library:

EXTEND.LIB

See also:

ACHOICE(), DBEDIT(), HARDCR(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT()

MEMOLINE()

Syntax:

MEMOLINE(<expC> [, <expN1>] [,<expN2>][, <expN3>] [,<expL>])

Purpose:

To extract a formatted line of text from a character string or memo field.

Arguments:

<expC> is the memo field or character string from which to extract lines of text.

<expN1> is the number of characters per line. The default is 79, the maximum is 254, and the minimum is four.

<expN2> is the line number to extract. The default is one.

<expN3> is the tab size. The default is four. If <expN3> is greater than or equal to <expN1>, then the tab size is <expN1> - 1.

<expL> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. The default is true (.T.).

Note that most arguments are optional. To skip an argument and specify further arguments, pass a dummy argument for the argument you wish to skip.

Returns:

A character string.

MEMOLINE() returns the line specifed by <expN2> in <expC> based on the the number of characters per line (<expN1>), the tab size (<expN3>), and wrapping behavior (<expL>).

If <expL> is true (.T.) and an end-of-line position breaks a word, it is word-wrapped to the next line. The next line then begins with the next non-blank character. If <expL> is false (.F.), MEMOLINE() returns the number of characters specified by the line width (<expN1>). The next line begins with the character following the next hard carriage return. This means that the intervening characters are truncated.

If the line has fewer characters than the indicated width, it is padded with blanks. If the line number is greater than the total number of lines in the expression, MEMOLINE() returns a null string ("").

Usage:

MEMOLINE() is designed to be used in combination with MLCOUNT() to extract lines of text from character strings and memo fields based on the number of characters per line. The basic method of operation is to determine the number of lines in the memo field or character string using MLCOUNT(), the number of characters per line, the tab size, and the wrapping behavior. Then navigate through the memo or character string with FOR...NEXT using the result of MLCOUNT() as the upper boundary of the loop. Within the loop, extract the current line with MEMOLINE() using the same values for the characters per line, tab size, and word wrap arguments.

Example:

The following example prints a memo field (Notes) which starts on row 12, at column 10, and is 40 characters wide.

```
row   = 0
col   = 10
width = 40
tab   = 3
wrap  = .T.
*
SET DEVICE TO PRINT
line_cnt = MLCOUNT(Notes, width, tab, wrap)
FOR curr_line = 1 TO line_cnt
   print_line = MEMOLINE(Notes, width, curr_line, ;
      tab, wrap)
   @ row + curr_line - 1, col SAY print_line
NEXT
SET DEVICE TO SCREEN
```

Library:

EXTEND.LIB

See also:

HARDCR(), MEMOEDIT(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT(), MLPOS()

MEMOREAD()

Syntax:

MEMOREAD(<expC>)

Purpose:

To read the contents of a text file from disk.

Argument:

<expC> is the name of the file you want to read from disk. Note that it must include an extension if there is one and can optionally include the path.

Returns:

A character string.

MEMOREAD() returns the contents of a text file as a character string. The maximum file size that can be read is 65,535 characters (64K), the maximum size of a character memory variable.

Examples:

The following uses MEMOREAD() to assign the contents of text file to both a memo field and a character variable.

```
* Notes is a memo field.
REPLACE Notes WITH MEMOREAD("Temp.txt")
charvar = MEMOREAD("Temp.txt")
```

This simple program uses MEMOREAD() to read a file from disk, edit it, and write it back.

```
* Editor.prg
PARAMETERS file
MEMOWRIT(file, MEMOEDIT(MEMOREAD(file)))
RETURN
```

Library:

EXTEND.LIB

See also:

REPLACE, HARDCR(), MEMOEDIT(), MEMOLINE(), MEMOTRAN(), MEMOWRIT(), MLCOUNT()

MEMORY()

Syntax: MEMORY(<expN>)

Purpose: To determine the amount of available memory.

Argument: <expN> is a numeric expression that must evaluate to zero.

Returns: An integer numeric value.

MEMORY(0) returns the free pool space for data manipulation.

Usage: See Chapter 9, *The Runtime Environment*

Library: CLIPPER.LIB

MEMOTRAN()

Syntax:

MEMOTRAN(<expC1> [,<expC2> [,<expC3>]])

Purpose:

To replace carriage return/line feed pairs.

Arguments:

<expC1> is the character string or memo field to replace soft or hard carriage returns in.

<expC2> is the character you specify to replace a hard carriage return/line feed pair.

<expC3> is the character you specify to replace a soft carriage return/line feed pair.

Returns:

A character string.

If you do not specify <expC2> and <expC3>, MEMOTRAN() replaces all hard carriage returns with semicolons, all soft carriage returns with spaces, and eliminates all line feeds.

Usage:

The default replacement values accommodate the requirements of the REPORT FORM command.

If you invoke a REPORT FORM that uses MEMOTRAN() and do not use MEMOTRAN() anywhere else in the program, be sure to declare MEMOTRAN() to the linker with EXTERNAL.

Example:

To strip all formatting characters from a memo field.

```
REPLACE Notes WITH MEMOTRAN(Notes, " ", " ")
```

Library:

EXTEND.LIB

See also:

REPLACE, HARDCR(), MEMOEDIT(), MEMOLINE(), MEMOREAD(), MEMOWRIT(), MLCOUNT()

MEMOWRIT()

Syntax:

MEMOWRIT(<expC1>, <expC2>)

Purpose:

To write a character string to a specified disk file.

Arguments:

<expC1> is the disk file name. Note that it must include the file extension if there is one and can optionally include the path.

<expC2> is the character string to write as the file contents.

Returns:

A logical value.

MEMOWRIT() returns true (.T.) if the writing operation is successful.

Example:

```
status = MEMOWRIT("Temp.txt", Notes)
IF status
   RUN Editor Temp.txt
ELSE
   ? "Write error"
ENDIF
```

Library:

EXTEND.LIB

See also:

HARDCR(), MEMOEDIT(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MLCOUNT()

MIN()

Syntax:

MIN(<exp1>, <exp2>)

Purpose:

To determine the smaller of the two numeric or date expressions.

Arguments:

<exp1> is the first numeric or date expression to be compared.

<exp2> is the second numeric or date expression to be compared.

Note that both arguments must be the same data type.

Returns:

A numeric or date value.

Usage:

MIN() is useful to insure the value of an expression is smaller than a specified maximum.

Examples:

```
a = 99
b = 100
? MIN(a, b)                          && Result: 99
? DATE()                             && Result: 09/01/87
? MIN(DATE(), DATE() + 30)           && Result: 09/01/87
```

Library:

CLIPPER.LIB

See also:

MAX()

MLCOUNT()

Syntax:

MLCOUNT(<expC>[, <expN1>][,<expN2>][,<expL>])

Purpose:

To count the number of word-wrapped lines in a character string or a memo field.

Arguments:

<expC> is the character string or memo field.

<expN1> is the number of characters per line. The default is 79, the maximum is 254, and the minimum is four.

<expN2> is the tab size. The default is four. If <expN2> is greater than or equal to <expN1>, then the tab size is <expN1> - 1.

<expL> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. The default is true (.T.).

Note that most arguments are optional. To skip an argument and specify further arguments, pass a dummy argument for the argument you wish to skip.

Returns:

An integer numeric value.

MLCOUNT() returns the number of lines in <expC> based on the number of characters per line (<expN1>), the tab size (<expN2>), and wrapping behavior (<expL>).

If <expL> is true (.T.) and an end-of-line position breaks a word, it is word-wrapped to the next line and the next line begins with the next non-blank character. If <expL> is false (.F.), MLCOUNT() counts the number of characters specified by the line width (<expN1>) as the current line. The next line begins with the character following the next hard or carriage return. This means that the intervening characters are ignored.

Usage:

MLCOUNT() is primarily used with MEMOLINE() to format memo fields or long character strings for printing. To do this, first use MLCOUNT() to return the number of word-wrapped lines. Then, using MEMOLINE() to extract each line, loop

through the memo field one line at a time until there are no lines left.

See MEMOLINE() for an expanded discussion.

Example:

```
count = MLCOUNT(memo, 40)
? count
```

Library:

EXTEND.LIB

See also:

HARDCR(), MEMOEDIT(), MEMOLINE(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLPOS()

MLPOS()

Syntax:

MLPOS(<expC>,<expN1>,<expN2>)

Purpose:

To determine the position of a specified line number in a character string or memo field.

Arguments:

<expC> is the character string or memo field.

<expN1> is the number of characters per line.

<expN2> is the line number.

Returns:

An integer numeric value.

MLPOS() returns the position in <expC> of the specified line number. If <expN2> is greater than the number of lines in <expC>, MLPOS() returns LEN(<expC>).

Example:

This example loads a text file from disk and then finds the position of the fifth line of text given a line length of 40 characters:

```
string = MEMOREAD("Temp.txt")
loc = MLPOS(string, 40, 5)
```

Library:

EXTEND.LIB

See also:

HARDCR(), MEMOEDIT(), MEMOREAD(), MEMOTRAN(), MEMOWRIT(), MLCOUNT()

MONTH()

Syntax:

MONTH(<expD>)

Purpose:

To convert a date value to a number representing the month of the year.

Argument:

<expD> is a date value to convert.

Returns:

An integer numeric value.

MONTH() returns a number in the range of zero to 12. Specifying a null date returns zero.

Usage:

MONTH() is useful when you require a numeric month value during calculations for such things as periodic reports.

Examples:

```
? DATE()                        && Result: 09/01/87
? MONTH(DATE())                 && Result: 9
? MONTH(DATE()) + 1             && Result: 10
```

Library:

CLIPPER.LIB

See also:

CDOW(), DOW(), CMONTH(), DAY(), YEAR(), CTOD(), DTOC(), DTOS(), DATE()

NETERR()

Syntax:

NETERR()

Purpose:

To determine if a USE, USE...EXCLUSIVE, or APPEND BLANK has failed in a network environment.

Returns:

A logical value.

NETERR() returns true (.T.) if the commands in the following table fail.

Table 6-15 Commands that Cause NETERR() to Fail

Command	Cause
USE	USE EXCLUSIVE by another process
USE...EXCLUSIVE	USE EXCLUSIVE or USE by another process
APPEND BLANK	FLOCK() by another process or another APPEND BLANK attempt at the same time

Usage:

See Chapter 10, *Using Clipper with a Local Area Network*, for more information.

Example:

```
USE Customer
IF NETERR()
   ? "Customer file in use by another"
   WAIT
   RETURN
ENDIF
```

Library:

CLIPPER.LIB

See also:

USE, USE...EXCLUSIVE, APPEND BLANK, FLOCK(), RLOCK()

NETNAME()

Syntax: NETNAME()

Purpose: To determine the current workstation identification.

Returns: A character string.

NETNAME() returns the workstation identification as a character string 15 characters long. If the workstation identification was never set or the application is not operating under the IBM-PC Network, it returns a null string ("").

Example:

```
? NETNAME()
```

Library: CLIPPER.LIB

NEXTKEY()

Syntax:

NEXTKEY()

Purpose:

To read the next keystroke without removing it from the keyboard buffer.

Returns:

An integer numeric value.

NEXTKEY() returns a number from -39 to 386, identifying the ASCII code of the key pressed (the same value as returned by INKEY() and LASTKEY()). If the keyboard buffer is empty, NEXTKEY() returns zero.

NEXTKEY() returns values for all function, Alt-function, Ctrl-function, Alt-letter, and Ctrl-letter key combinations. See Appendix G for the complete list of key values.

Usage:

Since NEXTKEY() does not remove the key from the keyboard buffer, you can use it to poll the keyboard and then pass control to a routine that uses a wait state to fetch the key from the buffer.

Example:

```
KEYBOARD CHR(27)
? NEXTKEY(), LASTKEY()          && Result: 27 0
? INKEY(), LASTKEY()            && Result: 27 27
```

Library:

EXTEND.LIB

See also:

INKEY(), LASTKEY()

PCOL()

Syntax:

PCOL()

Purpose:

To determine the current column position of the print head.

Returns:

An integer numeric value.

PCOL() returns a numeric value representing the last column position printed plus one. For example, if you print a string five characters long beginning at column one, a subsequent PCOL() returns six.

An EJECT resets PCOL() to zero. If you need to reset the internal printer column value without performing an EJECT, use SETPRC().

Usage:

PCOL() helps keep track of printer column position when you want to place text on a line relative to other text on the line.

Example:

```
SET DEVICE TO PRINT
@ 10, 0 SAY "Hello,"
@ 10, PCOL() + 1 SAY "how are you?"
EJECT
SET DEVICE TO SCREEN
```

Results:

```
Hello, how are you?
```

Library:

CLIPPER.LIB

See also:

PROW(), SETPRC(), COL(), ROW()

PCOUNT()

Syntax:

PCOUNT()

Purpose:

To determine the number of actual parameters passed to a procedure or user-defined function.

Returns:

An integer numeric value.

PCOUNT() returns the number of parameters passed. If no parameters are passed, PCOUNT() returns zero. Note that PCOUNT() retains its value until you attempt to pass parameters to either a procedure or a user-defined function.

Examples:

```
PARAMETERS file
IF PCOUNT() = 0
   ACCEPT "File to use: " TO file
ENDIF
USE &file
```

Library:

CLIPPER.LIB

See also:

DO...WITH, PARAMETERS

PROCLINE()

Syntax:

PROCLINE()

Purpose:

PROCLINE() returns the current source code line number from the beginning of the current program file.

Returns:

An integer numeric value.

If you compile your program without source code line numbers (-l switch), the line number returned by PROCLINE() is unpredictable.

See Chapter 7, *Compiling and Linking* for more information.

Example:

```
? PROCLINE(), "memvar + ", memvar
```

Library:

CLIPPER.LIB

See also:

PROCNAME()

PROCNAME()

Syntax: PROCNAME()

Purpose: PROCNAME() returns the name of the current program or procedure being executed.

Returns: A character string.

Example:

```
? "Current procedure:", PROCNAME()
```

Library: CLIPPER.LIB

See also: PROCLINE()

PROW()

Syntax:

PROW()

Purpose:

To return the current row position of the print head.

Returns:

An integer numeric value.

EJECT resets PROW() to zero. If you need to reset the internal printer column value without performing an EJECT, use SETPRC(). Be aware that if you move the print head with CHR(10), Clipper is not aware of this and PROW() will not return the expected value.

Usage:

PROW() helps you keep track of the print head row position. It is useful when you want to place text on a line relative to another line on a page.

Example:

```
SET DEVICE TO PRINT
@ 10, 0 SAY "First string"
@ PROW() + 1, 0 SAY "Second string"
```

Result:

```
First string
Second string
```

Library:

CLIPPER.LIB

See also:

SET PRINT, SET DEVICE, PCOL(), COL(), ROW(), SETPRC()

RAT()

Syntax: RAT(<expC1>, <expC2>)

Purpose: To search a character string for the last instance of a specified substring and return the starting position as a numeric value.

Arguments: <expC1> is the character string to locate.

<expC2> is the character string to be searched.

Returns: An integer numeric value.

If the substring is contained within the target expression, RAT() returns the starting character position of the substring. If the substring is not found, RAT() returns zero.

Usage: RAT() is similar to the AT() function except that scanning begins from the right therefore locating the last instance of substring within a specified string.

Example:

```
fname = "C:\DBF\Sales.dbf"
? SUBSTR(fname,1, RAT("\", fname)) && Result: C:\DBF\
```

Library: EXTEND.LIB

See also: AT(), STRTRAN(), SUBSTR(), $

READEXIT()

Syntax:

READEXIT([<expL>])

Purpose:

To toggle the Uparrow and Dnarrow keys as READ exit keys.

Argument:

<expL> toggles the Uparrow or Dnarrow keys as exit keys for a READ. Setting READEXIT() true (.T.) activates them as exit keys; false (.F.) deactivates them. The default Clipper setting is false (.F.)

Returns:

A logical value.

READEXIT() returns the current setting prior to toggling to a new setting.

Examples:

```
var = SPACE(10)
READEXIT(.T.)
@ 10,10 SAY "Enter: " GET var
READ
READEXIT(.F.)
```

Library:

CLIPPER.LIB

See also:

@...SAY...GET, READ, READINSERT()

READINSERT()

Syntax: READINSERT([<expL>])

Purpose: To report the current insert mode setting for READ and MEMOEDIT() and optionally toggle it on or off.

Argument: <expL> toggles the insert mode on or off. True (.T.) turns insert on, while false (.F.) turns insert off. The default Clipper setting is false (.F.)

Returns: A logical value.

READINSERT() returns the current insert mode setting if an argument is not specified and the previous insert mode if the argument is specified.

Example: The following example sets the insert mode prior to entering MEMOEDIT() and resets the mode when MEMOEDIT() terminates:

```
ins_mode = READINSERT(.T.)     && Turn on insert mode.
x = MEMOEDIT(x)
READINSERT(ins_mode)           && Restore last setting.
```

Library: CLIPPER.LIB

See also: READ, MEMOEDIT(), READEXIT()

READVAR()

Syntax: READVAR()

Purpose: To determine the current GET/MENU variable.

Returns: A character string.

READVAR() returns the name (in upper case) of the current GET/MENU variable or a null string ("""), if none is pending.

Usage: READVAR() is used primarily for debugging purposes or in programs using SET KEY.

Example: In the following example, pressing F2 executes Getname and the name of the current GET variable displays on line 23:

```
a = SPACE(10)
b = SPACE(20)
SET KEY -1 TO Getname          && Set F2 to Getname.
@ 5,5 SAY "A:"   GET a
@ 6,5 SAY "B:"   GET b
READ
RETURN

PROCEDURE Getname
@ 23,00
@ 23,00 SAY "Getfield = " + READVAR()
INKEY(0)
@ 23,00
RETURN
```

Library: CLIPPER.LIB

See also: @...GET, READ, SET KEY

RECNO()

Syntax: RECNO()

Purpose: To determine the record number of the current work area.

Returns: An integer numeric value.

If a database file contains no records, RECNO() returns the value 1 and the BOF() and EOF() both return true (.T.).

If the record pointer is set to point past the last record in the file (by using a SKIP command to skip past the last record), RECNO() returns LASTREC() + 1 and EOF() returns true (.T.). If an attempt is made to set the record pointer before the first record in a file, RECNO() returns the value 1 and BOF() returns true (.T.).

Usage: RECNO() is useful when you want to determine a record number from within a program.

Example:

```
USE Customers
GO 3
record = RECNO()
? RECNO()                    && Result: 3
GO record
? RECNO()                    && Result: 3
GO TOP
? RECNO()                    && Result: 1
```

Library: CLIPPER.LIB

See also: BOF(), EOF()

RECSIZE()

Syntax:

RECSIZE()

Purpose:

To determine the record length of the current database file.

Returns:

An integer numeric value.

RECSIZE() returns the record length for the database file in USE in the current work area. If no database file is in USE, RECSIZE() returns zero.

Usage:

RECSIZE() determines the length of a record by taking the length of each field in the record and adding one character to the total (for the asterisk that indicates a deleted record). When this value is multiplied by LASTREC(), the product is the amount of space occupied by the file's records.

To determine the entire length of a file, add the length of the file header:

```
filesize = (RECSIZE() * LASTREC()) + HEADER()
```

RECSIZE() is useful in programs that perform automatic file backup. When used in conjunction with DISKSPACE(), the RECSIZE() function can assist in ensuring that sufficient free space exists on a disk before a file is stored.

Library:

EXTEND.LIB

See also:

DISKSPACE(), FIELD()/FIELDNAME(), HEADER(), LASTREC()/RECCOUNT()

RESTSCREEN()

Syntax:

RESTSCREEN(<expN1>, <expN2>, <expN3>, <expN4>, <expC>)

Purpose:

To display a previously saved screen region to a specified screen area.

Arguments:

<expN1...expN4> are the screen coordinates to display screen data contained in <expC>.

<expC> is a character string containing the screen data to display.

Returns:

There is no return value.

Usage:

RESTSCREEN() is used to redisplay a screen region saved with SAVESCREEN(). The screen location to restore may be the same or different. If you specify a new screen location, be sure the new screen region is the same size or you will get ambiguous results. In addition, do not use RESTORE SCREEN to restore screen regions saved with SAVESCREEN() or you will get equally ambiguous results.

Example:

```
winbuff = SAVESCREEN(1, 1, 20, 40)
<statements>...
RESTSCREEN(1, 1, 20, 40 winbuff)
```

Library:

EXTEND.LIB

See also:

RESTORE SCREEN, SAVE SCREEN, SAVESCREEN()

RIGHT()

Syntax:

RIGHT(<expC>, <expN>)

Purpose:

To extract a specified number of characters from a character string beginning with the rightmost character.

Arguments:

<expC> is the character string from which to extract characters.

<expN> is the number of characters to extract.

Returns:

A character string.

RIGHT() returns the rightmost <expN> characters of <expC>. If <expN> is negative or zero, RIGHT() returns a null string ("""). If <expN> is larger than the length of the character string, RIGHT() returns the entire string. The maximum string size is 65,535 (64K) bytes.

Usage:

Note that Right() is the same as SUBSTR() with a negative first argument. For example, RIGHT("ABC", 1) is the same as SUBSTR("ABC", -1)

Example:

```
? RIGHT("ABCDEF", 3)            && Result: DEF
? SUBSTR("ABCDEF", -3)          && Result: DEF
```

Library:

EXTEND.LIB

See also:

LEFT(), LTRIM(), RTRIM()/TRIM(), STUFF(), SUBSTR(), TRIM()

REPLICATE()

Syntax: REPLICATE(<expC>, <expN>)

Purpose: To repeat a character string a specified number of times.

Arguments: <expC> is the character string to repeat.

<expN> is the number of times to repeat <expC>.

Returns: A character string.

The maximum size of the string returned is 65,535 (64K) bytes. Specifying a zero as the numeric argument returns a null string.

Usage: REPLICATE() is useful anywhere you want to repeatedly display, print, or stuff the keyboard with one or more characters.

Example:

```
? REPLICATE("*", 5)          && Result: *****
? REPLICATE("Hi ", 2)        && Result: Hi Hi
? REPLICATE(CHR(42), 5)      && Result: *****
```

Library: CLIPPER.LIB

See also: SPACE()

RLOCK()/LOCK()

Syntax:

RLOCK()/LOCK()

Purpose:

To lock the current record in the current work area.

Returns:

A logical value.

RLOCK() returns true (.T.) if you successfully lock a record. Otherwise, it returns false (.F.).

Usage:

RLOCK() locks the current record in the selected work area. It remains until you lock another record, UNLOCK, CLOSE the DATABASE, or FLOCK().

Note that unlike dBASE III PLUS, RLOCK() does not lock other work areas in a relation chain.

For more information, refer to Chapter 10.

Example:

```
IF RLOCK()
   DELETE
ENDIF
```

Library:

CLIPPER.LIB

See also:

USE...EXCLUSIVE, SET EXCLUSIVE, UNLOCK, FLOCK()

ROUND()

Syntax:

ROUND(<expN1>, <expN2>)

Purpose:

To return a value rounded to the specified number of decimal places.

Arguments:

<expN1> is a numeric expression to be rounded.

<expN2> is the number of decimal places in the return value.

Returns:

A numeric value.

ROUND() rounds <expN1> to the number of places specified by <expN2>. Specifying a zero or negative value for <expN2> allows rounding of whole numbers. A negative <expN2> indicates the number of places to the left of the decimal point to round.

The display of the return value does not obey DECIMALS SETting unless SET FIXED is ON. With FIXED OFF, the display of the return value contains as many decimal digits as you specify for <expN2> or zero if <expN2> is less than one.

Note that Clipper rounds up in all instances. There is no odd/even rule.

Usage:

The ROUND() function is useful when you want to use a number with less precision than it currently has.

Examples:

```
SET DECIMALS TO 2
SET FIXED ON
? ROUND(10.10, 0)                  && Result: 10.00
? ROUND(10.49999999999999, 0)      && Result: 10.00
? ROUND(10.51, 0)                  && Result: 11.00
? ROUND(10.51, -2)                 && Result: 0.00
? ROUND(101.99, -1)                && Result: 100.00
? ROUND(109.99, -1)                && Result: 110.00
? ROUND(109.99, -2)                && Result: 100.00
```

Library:

CLIPPER.LIB

See also:

INT()

ROW()

Syntax:

ROW()

Purpose:

To return the current row position of the cursor on the screen.

Returns:

An integer numeric value.

ROW() returns a number in the range zero to 24 with zero the first row on the screen. When you CLEAR the screen, ROW() returns zero. After a READ terminates, ROW() returns 23.

Usage:

ROW() is useful when you use relative addressing within a program for cursor positioning. For example, the statement

```
@ ROW() + 5, 1 SAY SPACE(0)
```

positions the cursor five rows below its current location.

Examples:

```
@ 1, 1 SAY "this is line"
@ ROW(), COL() + 1 SAY "one"
@ ROW() + 1, 1 SAY "this is line two"
```

Results:

```
this is line one
this is line two
```

Library:

CLIPPER.LIB

See also:

@...SAY/GET, COL(), PCOL(), PROW()

SAVESCREEN()

Syntax:

SAVESCREEN(<expN1>, <expN2>, <expN3>, <expN4>)

Arguments:

<expN1...expN4> are the coordinates of the screen region to save.

Returns:

A character string.

SAVESCREEN() returns the specified screen region as a string up to 4000 bytes in length.

Usage:

SAVESCREEN() is used to save a screen region to a memory variable. To restore the partial screen, use RESTSCREEN().

Typically, you would save and restore a screen region when using a pop-up menu or for dragging a screen object.

Example:

The following user-defined function creates a pop-up menu using ACHOICE() in combination with SAVESCREEN() and RESTSCREEN(), returning the selection in the array of choices:

```
FUNCTION PopUp
PARAMETERS t, l, b, r, choices, ufunc, color
PRIVATE winbuff, menuchoice, lastcolor
*
* Save stuff.
winbuff = SAVESCREEN(t, l, b, r)
lastcolor = SETCOLOR(color)
*
* Clear region and display menu.
@ t, l TO b, r DOUBLE
menuchoice = ACHOICE(t + 1, l + 1, b - 1, r - 1,;
    choices, ufunc)
*
* Clean up and go home.
RESTSCREEN(t, l, b, r, winbuff)
SET COLOR TO (lastcolor)
RETURN menuchoice
```

Library:

EXTEND.LIB

See also:

RESTORE SCREEN, SAVE SCREEN, RESTSCREEN()

SCROLL()

Syntax:

SCROLL(<expN1>, <expN2>, <expN3>, <expN4>, <expN5>)

Purpose:

To designate a section of the screen to scroll up, down, or blank out.

Arguments:

<expN1> is the top of the window.

<expN2> is the left of the window.

<expN3> is the bottom of the window.

<expN4> is the right of the window.

<expN5> is the number of rows to scroll. A number greater than zero scrolls up the specified number of rows. A value less than zero scrolls down the specified number of rows and zero blanks the specified area.

Returns:

There is no return value.

Usage:

SCROLL() is used to emulate windows. This can be useful for defining a list in a specified part of the screen and allowing the user to scroll up when at the top of the window or down when at the bottom of the screen.

Examples:

```
@ 5, 10 SAY "Scroll Test"
INKEY(0)                && Wait until a key is pressed.
SCROLL(3,10,10,30,1)    && Scroll up 1 line.
INKEY(0)                && Wait until key is pressed.
SCROLL(3,10,10,30,-4)   && Scroll down 4 lines.
INKEY(0)                && Wait until key is pressed.
SCROLL(3,10,10,30,0)    && Blank out window.
```

Library:

EXTEND.LIB

See also:

@..CLEAR..TO, @..BOX, @..TO..[DOUBLE]

SECONDS()

Syntax:

SECONDS()

Purpose:

To determine the number of seconds elapsed since 12:00 AM.

Returns:

A numeric value.

SECONDS() returns the system time as "seconds.hundredths." The numeric value returned is the number of seconds elapsed since midnight, and is based on a twenty-four hour clock, in a range from 0 to 86399.

Usage:

The numeric value returned by SECONDS() provides a simple method of calculating elapsed time during the execution of a program.

Examples:

```
? TIME()                && Result: 10:00:00
? SECONDS()             && Result: 36000.00
*
start = SECONDS()
DO Process
elapsed = SECONDS() - start
? "Process took " + STR(elapsed) + " seconds"
```

Library:

CLIPPER.LIB

See also:

TIME()

SETCANCEL()

Syntax:

SETCANCEL([<expL>])

Purpose:

To toggle program termination with Alt-C, on or off.

Argument:

<expL> toggles the termination capability on or off. Specifying true (.T.) toggles termination on and false (.F.) toggles it off. The default setting when a Clipper application loads is true (.T.).

Returns:

A logical value.

SETCANCEL() returns the previous setting if an argument is specified; the current setting if an argument is not.

Usage:

SETCANCEL() serves two basic purposes: toggling the state of the termination key, Alt-C, and reporting the current or last state of SETCANCEL().

> **Note:** SET KEY takes precedence over Alt-C when the termination state is enabled.

> **Caution:** When SETCANCEL() has been set true (.T.), you cannot terminate a run-away program unless you have provided an alternative escape mechanism.

Example:

This example demonstrates how to provide an escape route with SETCANCEL() set to false (.F.). Note, however, that this method works only from within a wait state.

```
CLEAR
STORE SPACE(25) TO one, two
*
SETCANCEL(.F.)          && Disable Alt-C as halt key.
SET KEY 302 TO AltC     && Redefine Alt-C.
*
@ 10, 10 SAY "One:" GET one
@ 11, 10 SAY "Two:" GET two
READ
*
PROCEDURE AltC
PRIVATE temp_scr, answer
```

```
SAVE SCREEN TO temp_scr
*
lastcolor = SETCOLOR("W/B, N/G")
@ 6, 20 CLEAR TO 9, 58
@ 6, 20 TO 9, 58 DOUBLE
@ 7, 26 SAY "Alt-C: Do you want to quit?"
@ 8, 35 PROMPT " Yes "
@ 8, 41 PROMPT " No "
MENU TO answer
SET COLOR TO (lastcolor)
*
RESTORE SCREEN FROM temp_scr
*
IF answer = 1
   QUIT
ENDIF
*
RETURN
```

Library: CLIPPER.LIB

See also: SET ESCAPE, ALTD()

SETCOLOR()

Syntax:

SETCOLOR([<expC>])

Purpose:

To determine the current or previous color setting and optionally define colors for the next screen painting activity.

Arguments:

<expC> is a character string containing the standard, enhanced, border, background, and unselected color settings to make the current colors. Unlike SET COLOR TO, SETCOLOR() with no argument does not restore colors to their default values.

Note also that SETCOLOR() only supports color letter combinations and not color numbers.

Returns:

A character string.

SETCOLOR() returns a string representing the last color setting if <expC> is specified and the current setting if it is not specified.

Usage:

SETCOLOR() supports the same color settings as SET COLOR.

Standard/Enhanced: The "standard" and "enhanced" settings are color pairs with a foreground and an optional background color. "Standard" is used by all output, such as @...SAY and ?. "Enhanced" setting affects only the display of GETs.

Border: Border color is not supported on EGA or VGA monitors.

Background: The "background" is not currently supported.

Unselected: The "unselected" setting displays the current GET in the "enhanced" color while other GETs are displayed in the "unselected" color.

Attributes: High intensity and blinking are the supported attributes of colors. High intensity is denoted by "+" and blinking with "*." Each attribute specified is applied to the foreground color regardless of where it occurs in the setting definition.

Colors: The following table lists all the colors available:

Table 6-15.1 Clipper Color Table

Color	Letter
Black	N/Space
Blue	B
Green	G
Cyan	BG
Red	R
Magenta	RB
Brown	GR
White	W
Gray	N+
Yellow	GR+
Blank	X
Underline	U
Inverse Video	I

Examples:

```
* Get current color setting.
curr_color = SETCOLOR()

* Set a new color the same as SET COLOR.
SETCOLOR("BR+/N,R+/N")

* Get current color setting and set new one.
new_color = "BR+/N,R+/N"
old_color = SETCOLOR(new_color)

* Default colors.
SET COLOR TO
SETCOLOR()              && Result: W/N,I/N,N,N,I/N
```

Library: EXTEND.LIB

See also: SET COLOR TO

SELECT()

Syntax:

SELECT([<expC>])

Purpose:

To return the work area number of an alias.

Argument:

<expC> is the alias name whose work area number you want to return.

Returns:

An integer numeric value.

SELECT() returns a numeric value in the range from zero to 254. If <expC> is not specified, the number of the current work area is returned. If <expC> is specified and the alias does not exist, SELECT() returns zero.

Examples:

```
? SELECT()                    && Result: 1
SELECT 4
? SELECT()                    && Result: 4
*
USE Sales
SELECT 1
? SELECT("Sales")             && Result: 4
```

To reselect the value that was returned from the SELECT() function, use the SELECT command with the syntax: SELECT (<memvar>). For example:

```
SELECT 1
USE File1
last_area = SELECT()
SELECT 3
USE File2
SELECT (last_area)
? SELECT()                    && Result: 1
```

Library:

CLIPPER.LIB

See also:

SELECT, USE, ALIAS()

SETPRC()

Syntax: SETPRC(<expN1>, <expN2>)

Purpose: To SET the internal PROW() and PCOL() to the specified values.

Arguments: <expN1> is the new internal row position.

<expN2> is the new internal column position.

Returns: There is no return value.

Usage: This can be useful for sending a set up string to printers without changing where the program thinks the print head is positioned. In addition, this function can be used to suppress page ejects.

Example:

```
SET PRINT ON
row = PROW()
col = PCOL()
?? CHR(15)
*
SETPRC(row, col)
```

Library: CLIPPER.LIB

See also: SET DEVICE TO PRINT, PROW(), PCOL()

SOUNDEX()

Syntax:

SOUNDEX(<expC>)

Purpose:

To convert a character string to soundex form (phonetic complement) useful for INDEXing and searching.

Argument:

<expC> is the character string to convert to soundex form.

Returns:

A character string.

The string returned is a code in the form A9999.

Usage:

SOUNDEX() is useful for creating indexes and searching for strings where the precise spelling is unknown. SOUNDEX() is based on an algorithm by Donald E. Knuth.

Knuth, Donald E. (1973). Sorting and Searching. The Art of Computer Programming (Vol. 3), (p. 392). Reading, MA: Addison Wesley.

Example:

```
USE Sales
INDEX ON SOUNDEX(Salesman) TO Salesman
*
SEEK SOUNDEX("Bill")
? FOUND(), Salesman              && Result: .T. Bill

SEEK SOUNDEX("Billy")
? FOUND(), Salesman              && Result: .T. Bill
```

Library:

CLIPPER.LIB

Source:

EXAMPLEC.C

See also:

INDEX, LOCATE, SEEK, SET SOFTSEEK

SPACE()

Syntax:

SPACE(<expN>)

Purpose:

To return a string of spaces.

Arguments:

<expN> is the number of spaces to return up to a maximum of 65,535 (64K).

Returns:

A character string.

SPACE(0) returns a null string.

Usage:

SPACE() is a general purpose character function and so can be used for a number of different purposes including:

Initializing memory variables for data input.

```
m_cust = SPACE(LEN(Customer))
@ 10, 10 SAY "Customer Name " GET m_cust
READ
```

Formatting strings. The following example right-justifies a string within a column definition.

```
number    = "12345"
col_width = 12
? SPACE(col_width-LEN(LTRIM(number))) + LTRIM(number)
```

To create a function that places the cursor nondestructively.

```
FUNCTION Cursor
PARAMETERS row, col
@ row, col SAY SPACE(0)
RETURN  ''
```

Examples:

```
? LEN(SPACE(20))                    && Result: 20
? SPACE(5) + "Indented 5 spaces"
```

Result:

```
      String indented 5 spaces
```

Library:

CLIPPER.LIB

See also:

REPLICATE()

SQRT()

Syntax:

SQRT(<expN>)

Purpose:

To return the square root of the specified numeric expression.

Argument:

<expN> is a numeric expression from which a square root is to be returned. SQRT() operates on positive numbers only.

Returns:

A numeric value.

The value returned by SQRT() contains either the default number of decimal places (2) or the number of decimal places contained in the numeric expression, whichever is greater.

Usage:

The SQRT() function is useful in mathematical operations.

Examples:

```
SET DECIMALS TO 5
? SQRT(2)                    && Result: 1.41421
? SQRT(4).                   && Result: 2.00000
? SQRT(4) ** 2               && Result: 4.00000
? SQRT(2) ** 2               && Result: 2.00000
```

See also:

SET DECIMAL, SET FIXED

STR()

Syntax:

STR(<expN1> [,<expN2> [,<expN3>]])

Purpose:

To convert a numeric expression to a character string.

Arguments:

<expN1> is the numeric expression to convert to a character string.

<expN2> is the length of the character string to return including decimal digits, decimal point, and minus sign.

<expN3> is the number of decimal places to return.

Returns:

A character string.

If the optional length and decimal arguments are not specified, STR() returns 10 whole number digits including leading spaces, the number of decimals in the numeric expression for memory variables, and the entire contents of numeric fields including decimals digits.

If you specify <expN2> less than the number of whole number digits in <expN1>, STR() returns asterisks instead of the number.

If you specify <expN2> less than the number of decimal digits length required for the decimal portion of the returned string, Clipper rounds the number to the available number of decimal places.

If you specify <expN2> but omit <expN3> (no decimal places), the return value is rounded to an integer.

Date functions: the STR() of YEAR(), MONTH(), and DAY() returns different results than other numeric values. The STR() of MONTH() and DAY() returns a string of length three. The STR() of YEAR() returns a string five characters in length.

Usage:

STR() is useful when you want to display or print the results of numeric expressions.

Examples:

```
number = 123.45
? STR(number)                    && Result:        123.45
? STR(number, 4)                 && Result:  123
? STR(number, 2)                 && Result: **
? STR(number, 0)                 && Result: null string
? STR(number * 10, 7, 2)         && Result: 1234.50
? STR(number * 10, 12, 4)        && Result:    1234.5000
```

Library:

CLIPPER.LIB

See also:

SUBSTR(), VAL()

STUFF()

Syntax:

STUFF(<expC1>, <expN1>, <expN2>, <expC2>)

Purpose:

To delete, insert, and/or replace characters in a character string.

Arguments:

<expC1> is the target character string.

<expN1> is the starting position in the target string where the replacement occurs.

<expN2> is the number of characters to replace in the target string (<expC1>).

<expC2> is the replacement string.

Returns:

A character string.

Essentially STUFF() replaces <expN2> characters in the target string (<expC1>) beginning at <expN1> with <expC2>.

Usage:

With this basic structure STUFF() can perform the following six operations:

Insert: If you specify zero for <expN2>, no characters are removed from <expC1>, the replacement string (<expC2>) is inserted at <expN1>, and the entire string is returned.

Replace: If you specify a replacement string (<expC2>) the same length as <expN2>, <expC2> replaces characters beginning at <expN1>.

Delete: If replacement string (<expC2>) is a null string ("""), the number of characters specified by <expN2> are removed from <expC1> and the string is returned without any added characters.

Replace and insert: If the replacement string (<expC2>) is longer than <expN2>, all characters from <expN1> are replaced and the rest of the replacement string (<expC2>) is inserted.

Replace and delete: If the length of the replacement string (<expC2>) is less than <expN2>, all characters in the target string (<expC1>) are deleted from the end of <expC2> up to <expN2>.

Replace and delete rest: If <expN2> is greater than the length of the target string (<expC1>), the replacement string (<expC2>) is inserted at <expN1> and the rest of <expC1> is deleted.

Examples:

```
* Insert
? STUFF("ABCDEF", 2, 0, "xyz")    && Result: AxyzBCDEF

* Replace.
? STUFF("ABCDEF", 2, 3, "xyz")    && Result: AxyzEF

* Delete.
? STUFF("ABCDEF", 2, 2, "")       && Result: ADEF

* Replace and insert.
? STUFF("ABCDEF", 2, 1, "xyz")    && Result: AxyzCDEF

* Replace and delete.
? STUFF("ABCEDF", 2, 4, "xyz")    && Result: AxyzF

* Replace and delete rest.
? STUFF("ABCEDF", 2, 10, "xyz")   && Result: Axyz
```

Library: EXTEND.LIB

Source: EXAMPLEC.C

See also: AT(), LEFT(), RIGHT(), RAT(), STRTRAN(), SUBSTR()

STRTRAN()

Syntax:

STRTRAN(<expC1>, <expC2> [,<expC3>] [,<expN1>] [,<expN2>])

Purpose:

To search and replace within a character string.

Arguments:

<expC1> is the character string to search.

<expC2> is the sequence of characters to locate.

<expC3> is the sequence of characters to replace with. If this argument is not specified, all instances of the search argument are replaced with a null string ("").

<expN1> is the first occurrence that will be replaced. If this argument is omitted, the default is one.

<expN2> is the number of occurrences to replace. If this argument is not specified, the default is all.

Returns:

A character string.

Usage:

STRTRAN() performs a standard substring search within a character string. When it finds a match, it replaces the search string with the specified replacement string. All instances of the search string are replaced.

Note that STRTRAN() replaces substrings and therefore does not account for whole words. Additionally, SET EXACT ON has no effect on its operation. STRTRAN() always acts as if SET EXACT is OFF.

Examples:

```
string = "To be, or not to be"
? STRTRAN(string, "be", "compute")
```

Result:

```
To compute, or not to compute
```

Library:

CLIPPER.LIB

See also:

AT(), RAT(), SUBSTR(), $

SUBSTR()

Syntax:

SUBSTR(<expC>, <expN1> [,<expN2>])

Purpose:

To return a portion of a character string.

Arguments:

<expC> is the source character string. The maximum character string that you can take a substring of is 65,535 (64K) bytes, the maximum character string size in Clipper.

<expN1> is the starting position in the source string to begin the substring. If the starting position is positive, it is relative to the left most character in the string. If the starting position is negative, the starting position is relative to the right most character in the string.

<expN2> is the number of characters to return. If this argument is omitted, the substring begins the starting position and continues to the end of the string. If it is larger than the number of characters from the starting position to the end of the string, it is ignored.

Returns:

A character string.

Usage:

SUBSTR() is useful when you want to display or print only a portion of a character string.

Examples:

```
char = "this is a string"
*
? SUBSTR(char, 1, 4)          && Result: this
? SUBSTR(char, 6)             && Result: is a string
? SUBSTR(char, LEN(char)+2)   && Result: null string
? SUBSTR(char, -6)            && Result: string
? SUBSTR(char, -6, 3)         && Result: str
```

Library:

CLIPPER.LIB

See also:

AT(), RAT()

TONE()

Syntax:

TONE(<expN1>, <expN2>)

Purpose:

To sound a speaker tone for a specified frequency and duration.

Arguments:

<expN1> is the frequency of the tone to sound.

<expN2> is the duration of the tone measured in increments of 1/18 of a second. One second, therefore, is 18.

Note that for both arguments, non-integer digits are truncated.

Returns:

There is no return value.

Usage:

TONE() sounds the speaker at the specified frequency for the specified duration. The duration is measured in increments of 1/18 of a second. The frequency is measured in hertz (cycles per second). Frequencies less than 20 are inaudible. The following are the frequencies of standard musical notes:

Table 6-15.2 Table of Musical Notes

Pitch	Frequency	Pitch	Frequency
C	130.80	mid C	261.70
C#	138.60	C#	277.20
D	146.80	D	293.70
D#	155.60	D#	311.10
E	164.80	E	329.60
F	174.60	F	349.20
F#	185.00	F#	370.00
G	196.00	G	392.00
G#	207.70	G#	415.30
A	220.00	A	440.00
A#	233.10	A#	466.20
B	246.90	B	493.90
		C	523.30

Examples:

The following example is a beep procedure that sounds a tone sequence indicating a batch operation has completed:

```
PROCEDURE DoneBeep
*
TONE(150, 8)
TONE(130, 10)
RETURN
```

This example is a tone sequence you can use to indicate invalid keystrokes or boundary conditions:

```
PROCEDURE ErrorBeep
   TONE(300, 1)
   TONE(499, 5)
   TONE(700, 5)
RETURN
```

Library: EXTEND.LIB

Source: EXAMPLEA.ASM

See also: SET BELL

TIME()

Syntax:

TIME()

Purpose:

To return the system time.

Returns:

A character string.

The system time is returned in the format "hh:mm:ss," where "hh" is hours in 24-hour format, "mm" is minutes, and "ss" is seconds. Hours, minutes, and seconds are separated by colons.

Usage:

TIME() is useful when you want to include the time in a report or in a display.

Examples:

```
? TIME()                        && Result: 10:37:17
? SUBSTR(TIME(), 1, 2)          && Result: 10
? SUBSTR(TIME(), 4, 2)          && Result: 37
? SUBSTR(TIME(), 7, 2)          && Result: 17
```

Library:

CLIPPER.LIB

See also:

SECONDS(), DATE()

TRANSFORM()

Syntax:

TRANSFORM(<exp>, <expC>)

Purpose:

To format the results of an expression of any data type.

Arguments:

<exp> is the character, date, or numeric expression to format.

<expC> identifies the format of the returned character string.

Returns:

A character string.

TRANSFORM() takes the result of an expression of any data type and returns a formatted character string according to the template specified by <expC>. Note that since TRANSFORM() returns a character string, it does not solve the problem of formatting a totalled field in a REPORT FORM.

Usage:

TRANSFORM() allows you to format character, date, or numeric expressions in the same way you would with @...PICTURE converting each to a character string matching the specified format. The following is a complete list of functions and templates that TRANSFORM() supports. Refer to @...PICTURE for a complete discussion on how picture functions and templates work.

Table 6-16 TRANSFORM() Functions

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays date in SET DATE format
E	Displays date in British format
R	Non-template characters are inserted
X	Displays DB after negative numbers
Z	Displays zero as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to upper case

Table 6-17 TRANSFORM() Templates

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"
Y	Displays logicals as "Y" or "N"
!	Converts an alphabetic character to upper case
$	Display a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Specifies a decimal point position
,	Specifies a comma position

Examples:

The following example formats a number into a currency format using a template:

```
? TRANSFORM(123456, "$999,999")  && Result: $123,456
```

The following example formats a character string using a function:

```
? TRANSFORM("to upper", "@!")    && Result: TO UPPER
```

Library:

CLIPPER.LIB

See also:

@...PICTURE, LOWER(), STR(), UPPER()

TRIM()/RTRIM()

Syntax: TRIM(<expC>) / RTRIM(<expC>)

Purpose: To remove trailing spaces from the result of a character expression.

Argument: <expC> is the character string to remove the trailing spaces from.

Returns: A character string.

Usage: TRIM() is useful when you want to delete trailing spaces from a character string when concatenating with another string. For example, a typical application of TRIM() is the creation of formatted strings for names and addresses.

```
? TRIM(First) + IF(EMPTY(Mi), " ", Mi + ". ") + Last
? TRIM(City) + ", " + TRIM(State) + "  " + Zipcode
```

Test for null strings: you can use TRIM() as part of an expression to test for an empty character string. For example, you can use the condition, LEN(TRIM(<expC>)) = 0 or the more preferred method, EMPTY(<expC>).

Example:

```
string = "12345    "
? LEN(TRIM(string))                && Result: 5
*
string = SPACE(5)
? LEN(TRIM(string))                && Result: 0
```

Library: CLIPPER.LIB

See also: LTRIM(), SUBSTR()

TYPE()

Syntax:

TYPE(<expC>)

Purpose:

To return the type of the specified character expression.

Argument:

<expC> is a character expression whose content type will be determined. This can include a field, with or without the alias, a memory variable, or an expression of any type.

Returns:

TYPE() returns one of the following characters:

Table 6-18 TYPE() Return Values

Returns	Meaning
C	Character
D	Date
L	Logical
N	Numeric
M	Memo field
A	Array
U	Undefined
UE	Error syntactical
UI	Error indeterminate

Usage:

TYPE() returns the type of the specified expression. With its expanded capabilities, it can be used to test expression validity. There are, however, several special cases to note:

Array references: References to DECLAREd arrays return an "A." References to array elements return the type of the element. Invalid references return "U."

IF()/IIF(): In order to return the appropriate data type for an IF(), TYPE() evaluates the condition and then returns the type of the evaluated path. If either the IF() condition or the evaluated path are invalid, TYPE() returns "UE."

User-defined and EXTEND.LIB functions: If a reference is made anywhere in an expression to a function not found in

CLIPPER.LIB (a user-defined or EXTEND.LIB function), TYPE() returns a "UI."

Examples:

```
? TYPE("Alias_name-Fldvar")
? TYPE([SUBSTR("Hi There", 4, 5)])      && Result: C
? TYPE([UDF()])                         && Result: UI
? TYPE([IF(.T., "true", 12)])           && Result: C
```

Library:

CLIPPER.LIB

TYPE()

Syntax:

TYPE(<expC>)

Purpose:

To return the data type of the specified character expression.

Argument:

<expC> is an expression that evaluates to an expression to type. This can include a field (including the alias), a memory variable, or an expression of any type.

Returns:

TYPE() returns one of the following characters:

Table 6-18 TYPE() Return Values

Returns	Meaning
U	Undefined Variable has no value Array reference User-defined function IF()/IIF()
C	Character
D	Date
L	Logical
N	Numeric
M	Memo field
A	Array

Examples:

```
? TYPE("Alias_name->Fldvar")
? TYPE("SUBSTR([Hi There], 4, 5)")   && Result: C
? TYPE("UDF()")                      && Result: U
? TYPE("IF(5  1, [true], [false])    && Result: U
```

Library:

CLIPPER.LIB

UPDATED()

Syntax:

UPDATED()

Purpose:

To determine if a change was made to any of the pending GETs during the last or current READ.

Returns:

A logical value.

UPDATED() returns true (.T.) if data in a GET is added or changed.

Usage:

Each time READ executes, it resets UPDATED() to false (.F.). Note that within a READ, a change to any GET sets UPDATED() to true (.T.). Subsequent testing of UPDATED() within a SET KEY procedure or VALID reflects this status.

Examples:

```
USE Accounts
m_id = id
@ 1,0 SAY "Enter new ID";
      GET m_id
READ
IF UPDATED()
   REPLACE Id WITH m_id
ENDIF
```

Library:

CLIPPER.LIB

See also:

@...GET, READ

USED()

Syntax:

USED()

Purpose:

To determine if a database file is in USE in the current work area.

Returns:

A logical value.

USED() returns true (.T.) if there is a database file in USE in the current work area, false (.F.) if not.

Example:

```
SET EXCLUSIVE OFF
SELECT 1
USE Dbf1
IF USED()
   SET INDEX TO Ntx1
ELSE
   ? "Unable to USE file..."
   INKEY(0)
ENDIF
```

Library:

CLIPPER.LIB

See also:

USE, SELECT, NETERR(), SELECT()

USED()

Syntax:

USED()

Purpose:

To determine if a database file is in USE in the current work area.

Returns:

A logical value.

USED() returns true (.T.) if there is a database file in USE in the current work area, false (.F.) if not.

Example:

```
SET EXCLUSIVE OFF
SELECT 1
USE Dbf1
IF USED()
   SET INDEX TO Ntx1
ELSE
   ? "Unable to USE file..."
   INKEY(0)
ENDIF
```

Library:

CLIPPER.LIB

See also:

USE, SELECT, NETERR(), SELECT()

UPPER()

Syntax:

UPPER(<expC>)

Purpose:

To convert lower case alphabetic characters to upper case.

Argument:

<expC> is the character string to convert.

Usage:

UPPER() has a number of different uses. A common use is to compare strings where the specific instance of casing is not known. For example:

```
message = "Greetings"
? UPPER("greetings") = UPPER(message)
```

UPPER() is also useful for creating case independent index key expressions. For example:

```
USE Customers
INDEX ON UPPER(Last) TO Customers
*
* Later, use the same expression to lookup Customers.
mlast = SPACE(15)
@ 10, 10 GET mlast
SEEK UPPER(mlast)
```

Examples:

```
? UPPER("a string")        && Result: A STRING
? UPPER("123 char = <>")   && Result: 123 CHAR = <>
```

Library:

CLIPPER.LIB

See also:

LOWER()

VAL()

Syntax: VAL(<expC>)

Purpose: To convert numbers contained in a character expression to a numeric value.

Argument: <expC> is a character expression to convert.

Returns: A numeric value.

VAL() stops evaluating a character expression when a second decimal point, the first non-numeric character, or the end of the expression is reached.

Usage: VAL() is useful when you want to do calculations using numbers contained in a character expression.

Examples:

```
SET DECIMALS TO 2
SET FIXED ON
? VAL("12.1234")          && Result: 12.12
? VAL("12.1256")          && Result: 12.13
? VAL("12A12")            && Result: 12.00
? VAL("A1212")            && Result: 0.00
? VAL(SPACE(0))           && Result: 0.00
? VAL(SPACE(1))           && Result: 0.00
? VAL(" 12.12")           && Result: 12.12
? VAL("12 .12")           && Result: 12.00
```

Library: CLIPPER.LIB

See also: SET DECIMALS, STORE, STR(), SUBSTR(), TYPE()

WORD()

Syntax:

WORD(<expN>)

Purpose:

To convert numeric parameters of the CALL command from type DOUBLE to type INT.

Argument:

<expN> is the numeric value to covert to type INT.

Usage:

Use WORD() when you want to convert numeric parameters from DOUBLE to INT in order to reduce the overhead of the CALLed routine. Be sure that the value does not exceed positive or negative 32,767.

Example:

```
CALL C_proc WITH WORD(30000)
```

Library:

CLIPPER.LIB

See also:

CALL

YEAR()

Syntax: YEAR(<expD>)

Purpose: To convert a date value to a number representing the year.

Argument: <expD> is a date value to convert.

Returns: An integer numeric value.

YEAR() returns the year of the specified date value including the century digits.

Usage: YEAR() is useful when you want to do date arithmetic.

Examples:

```
? DATE()                    && Result: 09/01/87
? YEAR(DATE())              && Result: 1987
? YEAR(DATE()) + 14         && Result: 2001
```

Library: CLIPPER.LIB

See also: SET CENTURY, CDOW(), DOW(), CMONTH(), MONTH(), DAY(), CTOD(), DTOC(), DTOS(), DATE()

7 Compiling and Linking Your Programs

Chapter 7 contains the instructions for compiling and linking your Clipper application programs. Topics covered include:

- What does the Clipper compiler do?
- Compiling your programs.
- What does the Linker do?
- Linking your programs with the DOS Linker.
- Linking your programs with the PLINK86-Plus Linker.
- Batch Files.
- Linking programs with the Clipper Debugger.
- Linking programs with optional screen drivers.
- Linking programs with other function libraries.
- Overlays.

This chapter assumes that Clipper has been loaded into a directory named "Clipper" and that you have included Clipper in the PATH command of your AUTOEXEC.BAT file. This allows you to work and develop your programs in directories other than the "Clipper" directory. For more information on setting the DOS environment, see Chapter 9.

WHAT DOES THE CLIPPER COMPILER DO?

The Clipper compiler converts source code programs written in the dBASE III PLUS programming language into object code files. The source code programs are identified by the DOS filename extension ".PRG". Object code files are identified by the DOS filename extension ".OBJ". These object files can be linked with a Linker to produce an executable file, identified by the DOS filename extension ".EXE".

Clipper permits you to compile program, procedure, user-defined function, and format files (.PRG and .FMT) separately or in groups. The way in which you compile your programs will depend on how you have structured your application and how you wish to structure your executable file.

You can create your program, procedure, user-defined function, and format files using a line editor, text editor, or a word processor. You can also create database, report, and label files using the utility programs provided with Clipper.

COMPILING YOUR PROGRAMS

Running The Clipper Compiler

To run the Clipper compiler, enter the following Clipper command:

C>CLIPPER <[d:][\path\] filename>

where:

"CLIPPER" invokes the Clipper compiler.
"d:" is the disk drive identification letter.
"Path" is the DOS path designation to the specified directory.
"Filename" is the name of your program file. Do not include the .PRG file extension.

Note: In practice, it is more common to compile in the same directory in which your .PRG files are contained. Because it is not common to include the drive and path designations, the following examples will only specify a filename.

In the current directory, this command produces a single object file, with the same name as your program but with an .OBJ extension.

All of the program and format procedure files referenced in the program file are also compiled and become a part of the object file. For example, if the main program calls two other program files, all three of them are compiled.

Clipper must be able to find all program, format, and procedure files referenced in your application in the current directory or it returns, "cannot open, assumed external." If you are using a .CLP file to compile (see "Compiling A List of Programs Using A .CLP File" below), place the names of all program, format, and procedure files used by your application into your .CLP file.

If you are not using a .CLP file, Clipper searches for referenced files in a number of places. It first assumes that the files are within your code. If not, Clipper looks for a SET PROCEDURE TO <otherfile> command and searches the procedure file for the referenced files. Finally, Clipper will search your hard disk. If still not found, Clipper returns the "cannot open, assumed external" error message.

It is best to place all of the required program and procedure files to compile at one time in the same directory and then execute Clipper from that directory.

If an error occurs during compilation, an error message is displayed on the screen. A description of these error messages may be found in Appendix D. If you wish, you may use the DOS output redirection feature to redirect the displayed output of the compiler to a file that may be viewed or printed later. For example:

C>CLIPPER Yourprog > Errfile

This command directs the compiler output into a file called ERRFILE.

Compiler Options

The Clipper compiler provides seven optional switches that can be specified when you run Clipper. To include any or all of the options on the Clipper command line, use the following syntax:

C>CLIPPER <filename> [<-l> <-m> <-o> <-p> <-q> <-s> <-t>]

On the command line above, "filename" is the name of your program file. **Switches must be specified in lower case letters or the compiler ignores them**. The switches are indicated as follows:

Switch	Action
-l	no line numbers
-m	compile one file only
-o	move object file to path specifier
-p	pause for disk change
-q	suppress line numbers during compilation (quiet)
-s	syntax check only
-t	move temporary file to drive designator

No Line Numbers(-l)

The first option (-l) excludes the program's source code line numbers from the object file. This reduces the object code file by three bytes for each line of code in your program that contains a command.

> **Note:** The Clipper Debugger relies on line numbers to give you critical information regarding program errors. Therefore, this option should not be operative while using the Clipper Debugger.

Compiling a Single Program File (-m)

When you include this option, any program files or procedures called within the program file being compiled are not included in the resulting object file. This feature is useful when you want to make changes to one program file without having to re-compile the entire application.

Move Object File(-o)

The third option directs Clipper to create the object file in another directory. To specify this option, include the switch and the path specifier on the Clipper command line as follows:

C> CLIPPER <filename> -o <path specifier>

Pause(-p)

When you specify this option, the Clipper compiler loads and then pauses.

Suppress Line Numbers (-q)

This option suppresses line numbers from flashing on the screen while you are compiling your application. Line numbers in the source code are not affected.

Program Syntax Check(-s)

The sixth option suppresses the creation of an object file. Use this mode when you want to use the Clipper compiler to simply check your program for syntax errors.

Move Temporary File(-t)

The last option directs Clipper to create its temporary file on another drive. This option is especially useful with a RAM disk.

During the compilation process, Clipper creates a temporary file with a ".$$$" filename extension. To accelerate compile time, you may direct Clipper to write this file to a RAM disk.

To specify this option, use the following syntax:

C> CLIPPER <filename> -t <drive designator>

Compiling a List of Programs Using a .CLP File

Clipper provides a feature for your convenience that permits you to compile your program files, procedure files, format files and user-defined function files in groups creating a separate object file for each group.

Generally, you create your program, procedure, and user-defined function (.PRG) files for a particular application in a particular directory on your hard disk, or on a diskette that contains your entire application. Often a program will refer to other .PRG files. When you compile a program, Clipper will attempt to compile all of the .PRG files that are referenced in all of the programs.

There are several reasons that you may wish to compile a group of files into a single object file. For example, you may wish to compile a particular file or group of files for debugging purposes, or you may wish to compile certain groups of files for the purpose of creating overlays. (See the discussion below for information on creating overlays.)

Another situation indicates the need for .CLP files. Clipper, when compiling a program, establishes two tables: one containing the constants used by your program(s) and another containing the symbols. The number of constants and symbols that can be placed in these tables is limited. If your program exceeds the 64K Constants Table, Clipper will return the "Too Many Constants" error. Likewise, if your program exceeds the 64K Symbols Table, Clipper will return the "Too Many Symbols" error. You can resolve this by compiling some of the programs or procedure files separately using a .CLP file. Use the linker to combine these separate object files into one executable file.

Creating a .CLP File

You can create a .CLP file by using a line editor, text editor, word processor, or the DOS "COPY CON" command. The file that you create must have the extension ".CLP". List the names of the program and format files that you wish compiled in the .CLP file. Rename format files so that they show a ".PRG" extension in the directory (instead of the ".FMT" extension) before creating your .CLP file. You must NOT include the ".PRG" filename extension, however, when listing your files. Note that a .CLP file will not recognize the disk drive designator and path.

The following example creates a .CLP file using the DOS "COPY CON" command.

```
C>COPY CON:  MYPROG.CLP
Prog1
Prog2
Prog3
```

Press F6 and then Return.

This example creates the file MYPROG.CLP which contains the names of three .PRG files: PROG1, PROG2, and PROG3. For more information on using the "COPY CON" command, see your DOS manual.

Compiling a .CLP File

To compile the programs listed in the .CLP file, enter the following command:

```
C>CLIPPER @<filename>
```

where the "@" symbol identifies the following filename as a file that contains programs to be compiled. "Filename" is the name

of the .CLP file you have created. (DO NOT include the .CLP filename extension.) The above command produces a single object file with the name of the .CLP file and an .OBJ extension. For example:

C> CLIPPER @MYPROG

produces MYPROG.OBJ, assuming no errors are found.

WHAT DOES THE LINKER DO?

The primary purpose of the linker is to create an executable file, from one or more object files, that will run under DOS.

Object files are identified by the extension ".OBJ". Run-time library files, which contain information and routines that are used during execution, are identified by the extension ".LIB". Executable files are identified by the DOS filename extension ".EXE".

The DOS Linker can be used to link your program object files if overlays are not required. This linker is somewhat faster than PLINK86-Plus.

However, the PLINK86-Plus Linker, provided with Clipper, can produce overlays. (For more on overlays, see the discussion below.) This feature is very useful for segmenting the executable file when you have large application programs.

The executable file resulting from the use of either linker will run equally well on your computer.

LINKING YOUR PROGRAMS WITH THE DOS LINKER

Complete instructions for the DOS Linker can be found in your DOS manual. A sample of a DOS Linker session is included for your convenience.

> **Note:** Overlays cannot be constructed with the DOS Linker. To construct overlays, use PLINK86-Plus.

To run the DOS Linker interactively, enter the following command:

C>LINK

The following prompts then appear:

Object Modules[.OBJ]:[<filename>] + [<filename>] + ... + [<filename>]

Enter the filename(s) of your object file(s) and press Return.

Run File[filename.EXE]:<filename>

Enter the filename you have chosen for your executable file. If the filename is the same as your object file, just press Return.

List File[NUL.MAP]:<filename>

If you would like a MAP file generated, enter the filename and press Return; otherwise, just press Return.

Libraries[.LIB]:<filename>

Enter the library filenames you will link with your application (such as CLIPPER or EXTEND) and press Return.

You may also link your object files with a single command line. For example, the linker instructions below produce an executable file named TEST.EXE.

```
C> LINK TEST + PROG1 +
PROG2,,,\CLIPPER\CLIPPER/se:256
```

where:

"TEST, PROG1, and PROG2" are object files.
"\CLIPPER\CLIPPER" is the CLIPPER.LIB library.

The /SEGMENTS (/se) option which processes no more than the given number of segments per program. The number can be any integer from 1 to 1024. The option is used to override the default limit of 128 segments.

LINKING YOUR PROGRAMS WITH PLINK86-PLUS

There are three methods that can be used to run PLINK86-Plus: interactively, with a command line, and with a .LNK file. If an error occurs during the linking process, an error message or warning is displayed on the screen. A description of these warnings and error messages may be found in Appendix E. For a complete list of PLINK86-Plus commands, see Appendix I.

Warning: The first object file you link, in any of the three methods below, must be Clipper-compiled. Any others compiled by another compiler must be listed secondarily. If you list a non-Clipper compiled object file first, PLINK86-Plus will return, "Fatal error 67; Files are not Nantucket format, can't link."

Interactive Method

PLINK86-Plus can accept input interactively. To use this method just enter "PLINK86" and press Return. The linker will load and the following command prompt appears:

```
=>
```

You may enter a single command at the prompt and press Return, or you may enter several commands on the same line and then press Return. Each time you press Return, the command prompt will appear on the next line awaiting your entry.

To end the linking session, type a semicolon and press Return.

For example, the linker instructions below produce an executable file named TEST.EXE.

```
=> FILE Test, Prog1, Prog2
=> LIBRARY Clipper, Extend
=> ;
```

where:

"FILE", abbreviated FI, tells the linker what object file(s) to link.

"LIBRARY" tells the linker what libraries to reference.

In practice, it is not common to use the interactive input method. If you use this method, you will not be able to use the DOS output redirection feature.

Command Line Method

The simplest way to run the linker is to identify the required information on the linker command line and press Return.

Use the syntax below to produce an executable file:

C>PLINK86 FI <[d:][\path\] filename> <,filename...> <filename> LIB <[d:][\path\] library name>

where:

"PLINK86" invokes the PLINK86-Plus linker.

"FILE", abbreviated FI, tells the linker that the name(s) of the object file(s) follow.

"d:" is the disk drive identification letter.

"Path" is the DOS path designation to the specified directory.

"Filename" is the name of a compiled object file. You do not need to enter the .OBJ filename extension. More than one object file may be entered if they are separated by commas.

"LIBRARY", abbreviated LIB, tells the linker that the name(s) of the libraries follow.

"Libname" is the name of the required run-time library. By default PLINK86-Plus will look for CLIPPER.LIB and OVERLAY.LIB.

The file will be identified with the name of the first object file filename, and the extension ".EXE". For example, the command line below produces an executable file named TEST.EXE.

C> PLINK86 FI TEST, PROG1, PROG2, LIB CLIPPER, EXTEND

> **Note**: In practice, it is more common to link in the same directory in which your Clipper-compiled .OBJ files are contained. Because it is not common to include the drive and path designations, the following examples only specify a filename.

.LNK File Method

A file containing all or part of the statement input can be inserted in the command line with the following syntax:

C>PLINK86 @<filename>

where:

"@" is used to tell PLINK86-Plus to look in the specified file for the required commands.

"Filename" is the name of the file that contains the linker commands. A .LNK filename extension is assumed unless otherwise specified.

.LNK file input can be used at any point on the command line and can be input up to three levels deep, i.e., the first .LNK file can contain a reference to the second .LNK file, which can contain a reference to the third.

Example:

MYLIST.LNK contains:

```
OUTPUT Test.exe
FI Test @Mylist2.lnk
```

MYLIST2.LNK contains:

```
FI Test2
LIB Clipper, Extend
```

When you execute PLINK86-Plus by typing:

C> PLINK86 @Mylist

The result is the same as if you had typed:

```
OUTPUT Test.exe
FI Test.obj
FI Test2.obj
LIB Clipper, Extend
```

"OUTPUT" creates an executable file. The .EXE filename extension is assumed. If "OUTPUT" is not used, PLINK86-Plus generates an executable file using the primary name from the first input file.

"FILE", abbreviated FI, tells the linker that the name(s) of the object file(s) follow. Note that the .OBJ filename extension is optional.

"LIBRARY", abbreviated LIB, tells the linker that the name(s) of the libraries follow.

> **Note**: Be careful not to list the same object file in both .LNK files; otherwise, PLINK86-Plus will return "WARNING 11, Duplicate <filename>."

Using a .LNK file within the command line may be very useful. For example, the CL.BAT file (see *Using Batch Files* later in this chapter) may be modified as follows:

```
CLIPPER %1
IF NOT ERRORLEVEL 1 PLINK86 FI %1 @C:\CLIPPER\CL.LNK
```

where CL.LNK contains the LIB command and a list of commonly used libraries.

Several .LNK files may be included on the same command line:

```
C> PLINK86 @Link1 @Link2
```

.LNK files may also be mixed with other PLINK86-Plus commands. For example,

```
C> PLINK86 @Mylist VERBOSE
```

initiates the linking process and invokes the VERBOSE command.

VERBOSE displays a status line on the screen indicating the operations of the linker. This statement is useful if you are learning about PLINK86-Plus or if you are having difficulties linking your file; however, it slows the linking process.

For more information on .LNK files, see the discussion on overlays later in this chapter. See Appendix I for a complete description of PLINK86-Plus commands.

Running PLINK86-Plus

Any of the PLINK86-Plus commands may be used with any of the three linking methods described for executing PLINK86-Plus. When linking a complex application with overlays, however, the use of a .LNK file is recommended. The .LNK file is more efficient for multiple linking sessions and will maintain a record of the commands that were used.

If PLINK86-Plus cannot find the files specified in the FILE, LIBRARY or SEARCH statements, it checks the DOS environment for the DOS environmental variable "OBJ", typically initialized in the AUTOEXEC.BAT file. This variable usually refers to a directory that contains libraries and other commonly used object files. For example, assume that AUTOEXEC.BAT contains the following command:

SET OBJ=\CLIPPER

If PLINK86-Plus is searching for FILE.OBJ, it looks for \CLIPPER\FILE.OBJ. Note the absence of spaces around the equal sign.

If you wish, you may use the DOS output redirection feature by adding "> filename" to the end of the linker command line.

Without a .LNK file:

C>PLINK86 Fl Test > Linkerr

With a .LNK file:

C>PLINK86 @Mylist > Linkerr

At the end of a successful linking session, PLINK86-Plus will display a message like the following:

TEST.EXE (202 K)

This indicates that the output program TEST.EXE was successfully created. The number in parentheses is the executable file size. This size may be different from the actual size of the program on disk, especially when overlays are used. It also does not allow for the fact that the program will attempt to allocate more memory at execution time. Add 64-100K bytes to the executable file size and the result will be the minimum amount of memory required to run your application program. (Use the DOS CHKDSK command to ascertain the available memory.) See Chapter 9 for additional information on memory management.

For a discussion on compiling and linking with batch files, see a subsequent section in this chapter.

Tips for Using PLINK86-Plus

- A pound sign (#) may be used to place comments in the .LNK file. A percent sign (%) may be used in earlier versions of the linker.

- PLINK86-Plus is CASE SENSITIVE. It is a good idea to make all PLINK86-Plus commands upper case. Check your PLINK86-Plus link files for the following "gotchas." In all cases, adding LOWERCASE/UPPERCASE to the link file will solve the initial incompatibilities. (See Appendix I for more information.)

  1. OVERLAY command: "overlay prog, $constants" should be "OVERLAY PROG, $CONSTANTS". (See the discussion on overlays below.)

  2. A library with lower case symbols and an upper case index.

  3. Combining assembler with a case sensitive, high level language. Microsoft MASM forces all symbols to upper case. If the high level source code refers to those symbols in lower case, turn off case sensitivity with the LOWERCASE or the UPPERCASE command.

- This version of PLINK86-Plus is a special version for users of the Nantucket compiler. It provides all the major features of PLINK86-Plus, version 2.21.

If you receive a warning or error message and are unable to resolve the problem, contact Nantucket Support. (See Appendix A for details.) DO NOT contact Phoenix Technologies, Ltd.

USING BATCH FILES

Using the DOS "IF ERRORLEVEL" Command

As Clipper compiles a program and encounters errors, error messages are displayed. Clipper also tells DOS an error has occurred by returning a program exit code of "1". This program exit code can be detected by the DOS "IF [NOT] ERRORLEVEL" batch command. It can be used in a DOS batch file to prevent a program containing errors from being linked.

The following batch file commands work with the linker provided with the Microsoft languages.

```
CLIPPER %1
IF NOT ERRORLEVEL1 LINK %1
,,,\CLIPPER\CLIPPER/se:256
```

The following statements are contained in the "CL.BAT" file (supplied on your Clipper diskette) and work with the PLINK86-Plus linker.

```
CLIPPER %1
IF NOT ERRORLEVEL1 PLINK86 FI %1
```

When you enter the command:

```
C>CL Myprog
```

MYPROG.OBJ is linked only if Clipper finds no errors in the program.

Using a Batch Program to Compile and Link Your Applications (PLINK86-Plus)

Using a DOS batch file is a convenient way to compile and/or link your programs. The Clipper package includes two DOS batch programs to assist you. These batch programs are designed to automatically compile and link your application program. DO NOT include the filename extension ".PRG" when using a DOS batch file. The batch programs are described below.

CL.BAT

Used to compile and link your programs on your hard disk.

CLD.BAT

Used to compile and link your programs on a hard disk AND link in the Clipper Debugger, DEBUG.OBJ.

These batch programs are intended to assist you and to give an example of a processing procedure. You may modify these programs to suit your individual needs.

LINKING PROGRAMS WITH THE CLIPPER DEBUGGER

To include the Clipper Debugger in your executable file, you need only to identify the Clipper Debugger object file (DEBUG.OBJ) as one of the files to be linked. For example, enter the following command:

C>PLINK86 FILE Test,Debug

The Clipper Debugger will be included in your executable file (called TEST.EXE) and you will be able to use the Debugger commands when you run the program. (See Chapter 8 for a discussion on the Clipper Debugger.)

> **Warning:** Be sure that you specify your Clipper-compiled program first. If you do not, the link will be unsuccessful.

LINKING PROGRAMS WITH OPTIONAL SCREEN DRIVERS

Clipper writes directly to the screen based on IBM's screen memory addresses. Screen drivers are included on the Clipper disk to support other MS-DOS machines which are not 100 percent IBM compatible. One of these is:

ANSI.OBJ A generic driver used for other PC terminal compatibles that support the ANSI standard. This driver is limited and will not support cursor movement keys, function keys, and SAVE and RESTORE screens.

To include one of these drivers, identify it as one of the object files to be linked, as in the following example:

C>PLINK86 FILE Test,Ansi

> **Warning**: Be sure that you specify your Clipper-compiled program first. If you do not, the link will be unsuccessful.

When you include the ANSI object file in your executable file you must have the correct device driver present in the DOS environment. This is accomplished by including the DEVICE statement in your CONFIG.SYS file. (If you do not have a CONFIG.SYS file, you must create one.) If you are linking in the ANSI screen driver, enter the following statement:

DEVICE=ANSI.SYS

If any changes are made to the CONFIG.SYS file, you must turn the machine off and then back on again or re-boot so that the file is read and the changes are activated.

LINKING PROGRAMS WITH OTHER FUNCTION LIBRARIES

Many functions are contained in libraries other than the Clipper library. If you use these functions in your program, you must identify the library when linking. See Chapter 6 for the residing library of each function.

The example below links CLIPPER.LIB and EXTEND.LIB to the application MYPROG:

```
C> PLINK86 FI Myprog LIB Clipper,Extend
```

OVERLAYS

> **Note**: This discussion on overlays refers to PLINK86-Plus and .LNK files. For more information on these topics, refer to "Linking Your Programs with the PLINK86-Plus Linker" in this chapter. For a complete description of PLINK86-Plus commands, refer to Appendix I.

Unless you are certain that your application must contain overlays, attempt first to link and run your application without them. Using overlays unnecessarily (when your program does not exceed the memory capacity of your computer) will slow the execution of your program.

As a developer, you should be concerned about the size of the application you create. If your executable (.EXE) file is too large to fit within the available installed memory of a particular system, it will not run.

It is usually unacceptable to reduce the size of your application, assuming that you have used modular programming techniques. Installing more memory in the computer may solve the problem, but for large applications, even 640K may not provide enough available memory. Therefore, it may be desirable to create overlays.

What Are Overlays?

Overlaying is a technique that permits a large program to execute in a smaller amount of memory. Thus, with overlays, you may run programs much larger than the computer's memory can accommodate. Overlays are portions or segments of your application program that do not reside in your computer's RAM until the execution of that segment is required.

The main program code, or "root" portion of the program, is always resident in memory. The overlays are stored on disk until the root calls them into memory. The portion of memory that these overlays share is called the overlay area. An overlay is loaded whenever a routine in that overlay is called. One overlay overwrites another in the same memory space as each is required during program execution.

For example, suppose an overlay is loaded into memory and executed, then control returns to the root. When the root calls another overlay that is not currently in memory, the new overlay overwrites the first. Figure 1 illustrates how overlays appear in

memory. Notice that the largest overlay determines the amount of memory allocated for all of the overlays.

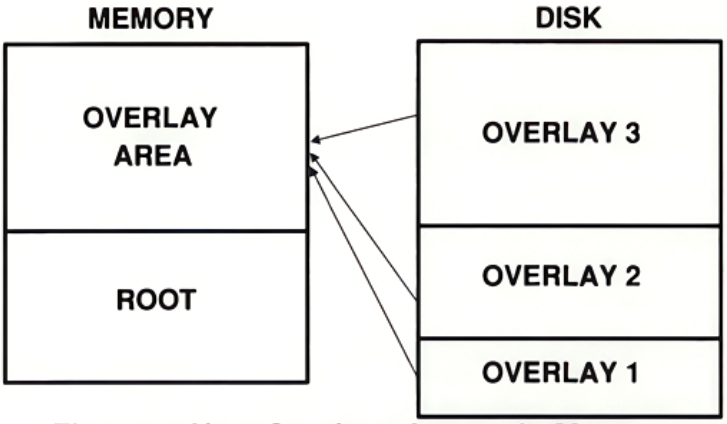


Figure 1: How Overlays Appear in Memory

A program containing a single overlay does not reduce the size of the program loaded into memory. The efficient use of memory space is achieved because many overlays are sharing one block of memory at different times.

You should never have to change your source code just to use overlays, assuming you have a modular design. The overlay structure is set up through the link commands. Overlays are then loaded automatically by PLINK86-Plus. If you decide to change your overlay structure, you need only change the link commands. Your source code remains intact.

Design Structure

When you write large applications with the intent of running them as a single executable program, you should be concerned with their logical structure. Most applications perform functional operations that can be divided into groups. These groups usually lend themselves to separate files that require little or no interaction between other files. This is the basis of an effective overlay structure.

The goal in overlay design is to create overlays containing functionally isolated sections of the program that execute independently of other sections. These sections, or "application segments," may contain one or more program, procedure, user-defined function, or format (.PRG or .FMT) files.

In anticipation of using overlays, divide your .PRG files into logical files. Figure 2 below illustrates a database management system. Eleven .PRG files have been compiled into six .OBJ files by compiling them with .CLP files. (For more on .CLP files, see "Compiling with .CLP Files" in this chapter.) The main program file, DB_MAIN, is the calling program for every other file. There can be no interaction between these other files.

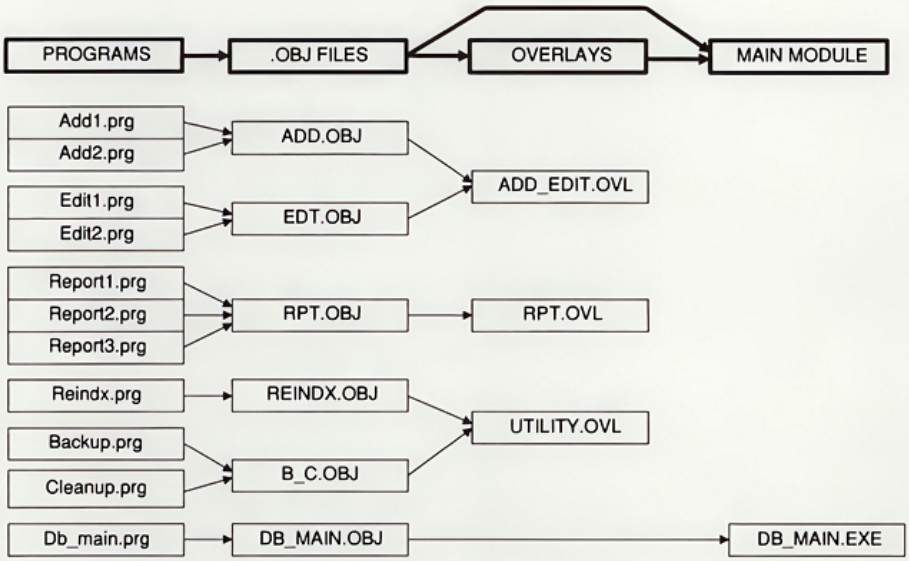


Figure 2: Design Structure Example

How To Create Overlays

Overlays are created with PLINK86-Plus commands at link time, in this case with a .LNK file. To link the object files created in Figure 2, the following .LNK file is used:

```
FI Db_main
LIB Extend
DEBUG
OVERLAY PROG, $CONSTANTS
BEGINAREA
   SECTION FILE Add, Edt
   SECTION FILE Rpt
   SECTION FILE Reindx, B_c
ENDAREA
```

Remember that, by default, PLINK86-Plus automatically looks for CLIPPER.LIB and OVERLAY.LIB.

When you link your programs, PLINK86-Plus segregates the information contained in the object files into classes. The data class contains "constant" information, such as string constants and memory variable names. The code class contains the actual program instructions. Clipper usually places the constants table in the main load file. To minimize the use of memory, the OVERLAY PROG, $CONSTANTS statement is used to place the data class information into the overlay rather than into the main file.

BEGINAREA defines the start of an overlay area. The end of the area is indicated by the ENDAREA command. SECTIONs created between the BEGINAREA and ENDAREA commands automatically become overlays that share the same memory space.

Overlays In Memory

Figure 3 illustrates how an overlay area created with the .LNK file above appears in memory. When the program is executed, the main program is loaded into memory and the space required by the largest overlay in the area is allocated. In Figure 3, the UTILITY overlay containing the REINDX and B_C object files is the largest of the three overlays; it determines the size of the overlay area allocated in memory. When it is loaded, it occupies the entire area. When another overlay is called, it overwrites the first. However, it does not occupy the entire allocated area. The remaining area is unused. One overlay procedure must never call a procedure in another overlay within the same overlay area since they are never present in memory at the same time.

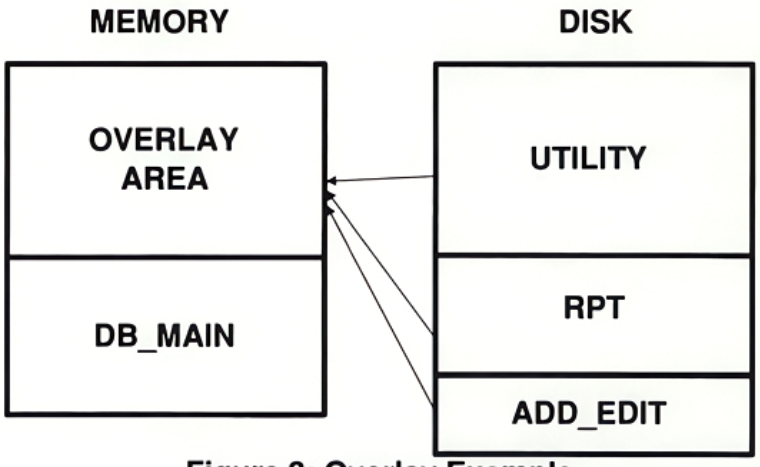


Figure 3: Overlay Example

Multiple Overlay Areas

The most basic overlay configuration consists of one root and one overlay area. Figures 1 and 3 illustrate this configuration. You may also create multiple overlay areas. For example, modify the .LNK file as follows:

```
BEGINAREA
    SECTION FILE Add, Edit
    SECTION FILE Rpt
ENDAREA
BEGINAREA
    SECTION FILE Reindx
    SECTION FILE B_c
ENDAREA
```

Figure 4 below illustrates the new configuration.

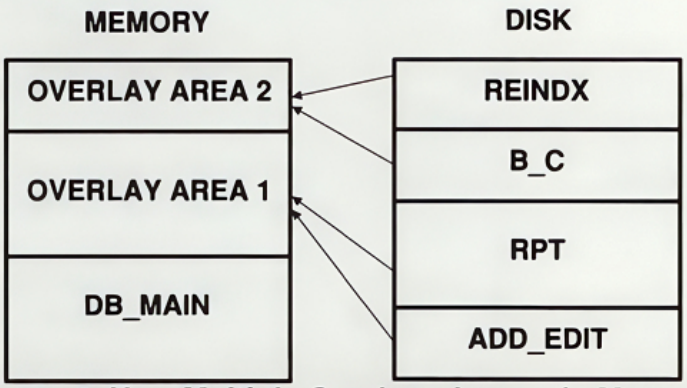


Figure 4: How Multiple Overlays Appear in Memory

Notice that the total memory space used by both Overlay Area 1 and Overlay Area 2 is less than the space used by the single overlay area illustrated in previous examples. This is because the UTILITY overlay, previously the largest overlay, was divided. RPT now becomes the largest overlay and dictates the size of Overlay Area 1.

Internal and External Overlays

You can create two kinds of overlays, internal and external, with PLINK86-Plus. Both types function identically. They differ only in that external overlays are contained in separate files, identified with the .OVL extension.

Internal overlays are created with the SECTION FILE statement. They are included in the .EXE file. Usually the main load file and all (internal) overlays are written to the same output file. The overlays appear at the end of the file and the operating system is

unaware of their presence when loading the program. Thus, with internal overlays, a directory search is not required to find the overlays files and the program executes faster than the same program with external overlays.

However, external overlays may be useful in several cases, such as when your application does not fit on a single floppy disk, or when you need to determine the actual size of the overlays. External overlays may be transferred on a disk other than the one on which the main program file resides. Overlay files need only be available to the program at execution time.

Overlays are placed in separate files with the SECTION statement's INTO option. To create external overlays rather than internal, modify the example .LNK file to reflect the following:

```
BEGINAREA
   SECTION INTO Add_edit.ovl FILE Add, Edit
   SECTION INTO Rpt.ovl FILE Rpt
   SECTION INTO Utility.ovl FILE Reindx, B_c
ENDAREA
```

Nested Overlays

You may also nest an overlay area within another overlay area. This technique, while completely permissible, requires very careful program and overlay structure planning. If managed improperly, overlays can play havoc with both your application and your sanity. Careful planning is imperative.

Figure 5 illustrates how a nested overlay appears in memory. Notice that Overlay Area 2 can only reside in memory when Overlay 3 is in memory.

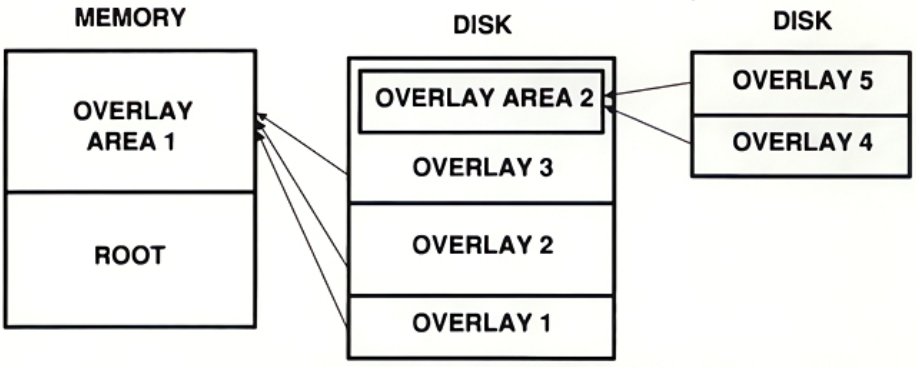


Figure 5: How Nested Overlays Appear in Memory

In the following examples, the database management system used in previous examples has been divided into a utility menu program with nested overlays for the REINDX and B_C (backup_cleanup) functions. The modifications are accomplished with the following .LNK file. The second BEGINAREA appears before the first ENDAREA, creating a nested overlay. Note that the .LNK file creates external overlays for later use with a map file.

```
FI Db_main
LIB Extend
DEBUG
OVERLAY PROG, $CONSTANTS
BEGINAREA
    SECTION INTO Add_edit.ovl FILE Add, Edit
    SECTION INTO Rpt.ovl FILE Rpt
    SECTION INTO Utilmenu.ovl FILE Utilmenu
    BEGINAREA
        SECTION INTO Utilmenu.ovl FILE Reindx
        SECTION INTO Utilmenu.ovl FILE B_c
    ENDAREA
ENDAREA
```

Optimally, a nested overlay does not increase the calling overlay to a size larger than the area within which it is contained. Figure 6 illustrates how the nested overlay created with the above .LNK file appears in memory.

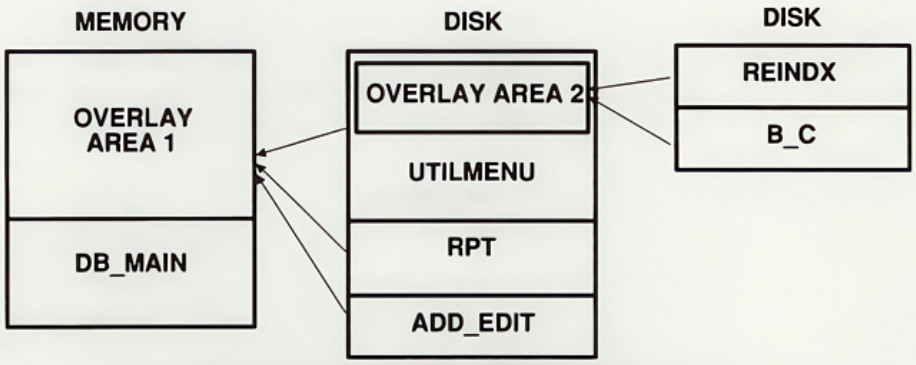


Figure 6: Nested Overlays Example

This approach saves the difference by which the UTILITY overlay is larger than the RPT overlay. However, a small amount of overhead is required for the memory used by the overlay table for two more overlays. In this case it saved memory; in other cases it might not have. The final judge is PLINK86-Plus, which tells you the load size at the completion of the linking process.

You may easily compare different overlay schemes by utilizing levels of .LNK files. .LNK file input can be used at any point on the link command line. It is useful to create one .LNK file (LINKA.LNK) containing the FILE, LIBRARY, and OVERLAY PROG, $CONSTANTS statements, and another .LNK file (LINKB.LNK) containing the BEGINAREA and ENDAREA statements, and those SECTION statements between that create the overlays. You may use the following command:

PLINK86 @Linka @Linkb

to compare different overlay schemes by interchanging LINKB.LNK with LINKC.LNK and so on.

Overlay Management

It is important to make sure all overlays in a given area are approximately the same size. Remember that the largest overlay dictates the amount of memory allocated for overlay execution. There are two ways to determine the load size of your overlays and the amount of memory any one area consumes. The first method is to use the MAP option with PLINK86-Plus and create a summary map. The second is to create external overlays and use the DOS directory.

PLINK86-Plus MAP Option

The PLINK86-Plus MAP statement generates memory map reports. A summary map is generated with the following syntax:

MAP = FILE.MAP S

where FILE.MAP is the filename and S is the flag argument used to select the desired report(s). A .MAP filename extension is used if none is specified. If a flag is not given, PLINK86-Plus will create MAP A. Please refer to Appendix I for additional flag descriptions.

The .LNK file is used to create the map:

```
FI Db_main
LIB Extend
DEBUG
OVERLAY PROG, $CONSTANTS
MAP = Db_main.map S
BEGINAREA
    SECTION INTO Add_edit.ovl FILE Add, Edit
    SECTION INTO Rpt.ovl FILE Rpt
    SECTION INTO Utility.ovl FILE Reindx, B_c
ENDAREA
```

The S option lists all program sections alphabetically with their assigned disk, memory addresses, and size. The MAP S column headings are listed below (reprinted by permission):

HEADING	DESCRIPTION
Maddr	Memory address where the section will be loaded.
Msize	Memory space used by the section.
Daddr	Address where the section is stored in a disk file.
Dsize	Disk space used by the section.
Lev	Level number of this section (0 = permanently resident).
Ov#	Overlay number of the section, where zero is the root section of the program that is loaded by the operating system. A non-zero indicates a section that will be loaded by the overlay loader.
Fth	Overlay number of the section's ancestor section within overlay structure (0 = no ancestors).
Pload	Preload flag. "Yes" indicates a section that will be loaded by the overlay loader before program execution actually begins.

An example MAP S generated by PLINK86-Plus is printed below. Note that the MAP output is in hex. A hex to decimal conversion chart follows.

PLINK86-Plus 2.21
Memory Map for C:DB_MAIN.EXE
Sections:

Name	Maddr	Msize	Daddr	Dsize	Lev	Ov#	Fth	Pload
Root	0	23370	2200	23370	0	0	0	No
(ADD_EDIT)	23370	ab60	60	ab60	1	1	0	No
(RPT)	23370	e9c0	60	e9c0	1	2	0	No
(UTILITY)	23370	f670	c0	f670	1	3	0	No
	329e0	a2f0	26680	a2f0	0	4	0	Yes

DOS Directory

You may determine the size of your overlays by listing the directory entries for all .OVL files after linking. The DOS directory appears below:

Directory of C:\DB

ADD_EDIT	OVL	43968
RPT	OVL	59936
UTILITY	OVL	63280

A Comparison

Because the MAP output speaks only hex, a hex to decimal chart is useful for comparing the PLINK86-Plus MAP and the DOS DIR. The decimal conversions for the above map appear below, followed by a converted excerpt of that map.

Hex	Decimal
60	96
C0	192
AB60	43872
E9C0	59840
F670	63088

Notice that the total of Dsize and Daddr from the map listing is equal to the file size from the directory.

File	Daddr	+	Dsize	=	Dtotal
ADD_EDIT	96		43872		43968
RPT	96		59840		59936
UTILITY	192		63088		63280

In order to make the program in this example use less memory it would be best to work with the files contained in the Utility overlay. Until the size falls below 59840, it controls the size of the overlay area. The actual amount of memory allocated to the overlay area here is 329E0 hex - 23370 hex which is F670 hex or 63088 bytes.

The map is more accurate. However, maps may be difficult to work with unless you have a hex to decimal calculator.

PLINK86-Plus Map - Nested Example

The PLINK86-Plus DEBUG command provides a way to test and debug overlays by identifying each overlay as it is loaded during execution. The overlay is identified by overlay number, determined sequentially by the order in which the overlays are defined in your .LNK file. When DEBUG is used with a link map, you may follow the overlay process at each step. The map indicates the overlay number under the heading "Ov#", as shown in the MAP S example below. Note that this is the map that was generated after nested overlays were created.

PLINK86plus 2.21
Memory Map for C:DB_MAIN.EXE
Sections:

Name	Maddr	Msize	Daddr	Dsize	Lev	Ov#	Fth	Pload
Root	0	23370	2200	23370	0	0	0	No
(ADD_EDIT)	23370	ab60	60	ab60	1	1	0	No
(RPT)	23370	e9c0	c0	e9c0	1	2	0	No
(UTILMENU)	23370	459	e0	459	1	3	0	No
(REINDX)	237c9	7528	539	7528	2	4	3	No
(B_C)	237c9	8238	7a61	8238	2	5	3	No
	329e0	a2f0	26680	a2fo	0	6	0	Yes

The nested overlays that were created are reflected under the "Lev" column by the appearance of a second level. Each level number is important because it corresponds with the number of "ancestors" the overlay has. Each ancestor must already reside in memory before the overlay can be loaded. The overlay's immediate ancestor is the last section defined prior to the beginning of the overlay area at a lower address. In the nested overlay example, the ancestor of REINDX is UTILMENU. The family of ancestors is the ancestor, the ancestor's ancestor...and that ancestor's ancestor, until a resident section is reached. Remember, the "root" portion of the program is always resident in memory. The ancestors of B_C are UTILMENU and DB_MAIN. DB_MAIN is also the resident section.

Resident sections are assigned a level number "0". The level number is increased by one when a BEGINAREA statement is entered, and decreased by one at the next ENDAREA statement. Level numbers will range from 0 to 31 because overlays can be nested up to 32 levels deep.

8 The Clipper Debugger

Chapter 8 contains information necessary for using the new Clipper Debugger. Topics covered include:

- An explanation of the Clipper Debugger
- Linking in the Debugger
- How to use the Clipper Debugger
- How to navigate within the Debugger
- An explanation of the menu bar selections

WHAT IS THE CLIPPER DEBUGGER?

The Clipper Debugger is designed to assist in finding program errors or "bugs." A menu bar and pull-down menus within the Debugger greatly facilitate the debugging process. With the Debugger linked to your application you have the choice of executing your program without intervention or one line at a time. Several options are offered to determine how your program is working. You can:

- Execute the program normally or step by step
- Display the status of the program environment (SET commands)
- Display the structure of active database files
- Display the value of an expression
- Display the index number and key expression
- Display the database structure
- Display related aliases and relation expression values
- Display a general overview of a database
- View the public and private memory variables and their current contents
- Create or change the contents of any private or public memory variable
- Establish breakpoints where you wish the program to stop
- Obtain context specific help at any time

In addition to these features, Clipper displays runtime errors when they occur during program execution. You may invoke the Debugger when a runtime error occurs.

LINKING IN THE DEBUGGER

The Clipper Debugger, DEBUG.OBJ, must be linked in to your application. Examples of linker commands that will include the Clipper Debugger in your application are shown below.

> Important: If you are currently using the "No Line Numbers" compiler switch (-l), the Clipper Debugger will NOT work. Do not use this option until you have debugged your application.

Using The PLINK86-Plus Linker

To link your program (TEST.OBJ is used for the examples below) and the Clipper Debugger with PLINK86-Plus, enter the following single command:

```
C>PLINK86 FI TEST, \CLIPPER\DEBUG LIB
     \CLIPPER\CLIPPER
```

The CLD.BAT batch file included on the Clipper disk automatically links in the Debugger.

> **Caution:** Identify your application object module first. If you identify the Debugger first, the resulting executable program will NOT work.

Using The DOS Linker

To link your program, TEST.OBJ, and the Clipper Debugger with the DOS linker, enter the following single command:

C>LINK TEST \CLIPPER\DEBUG,,,\CLIPPER\CLIPPER

USING THE CLIPPER DEBUGGER

When you run your application with the Debugger, DOS loads your program into memory and six menu selections are displayed across the top of your screen. These selections form a menu bar with some selections containing additional pull-down menus. The box shown on the right side of the menu bar is called a Watch box. By default the Watch Box displays the active procedure name to the right of "Proc", the current line number to the right of "Line", and the breakpoint system status. Additionally, up to 16 expressions can be set and tracked. This option is explained more fully in the Watch Menu section.

Invoking the Debugger

The Debugger is invoked by pressing Alt-D. When entering information into the Debugger, the Esc key cancels the input, while the Return key accepts the input.

ALTD() allows you to invoke the Debugger from within your application. The optional parameter <expN> determines the Debugger mode used when it is invoked either by the ALTD() function or by manually pressing the Alt-D keys. See Chapter 6, Clipper Functions, for more information on ALTD().

NAVIGATION WITHIN THE DEBUGGER

Menus

Use the arrow keys to move from one menu selection to another. Once within a pull-down menu, leftarrow and rightarrow keys allow you to move from one pull-down menu to another without returning to the menu bar first. Home and End always take you to the first and last menu selections respectively. If the highlighted word "WAITING" is on the

screen, press any key to return to the menu bar. Otherwise, the Esc key takes you up one level each time it is pressed.

The pull-down menus allow "first letter selection." You can make selections by typing in the first letter of your menu choice instead of using cursor movement keys. If more than one menu item begins with the same letter, use cursor movement keys to make your selection.

Additionally, the Debugger allows the use of speed keys for rapid movement through menus and options. See the Help Menu write up under MENU BAR SELECTIONS of this chapter for further information and a complete list of the speed keys and their purpose.

Procedure and Alias Lists

Displays which allow a selection of procedures or aliases have their own set of navigational rules which are detailed in the table below.

Table 8-1 Navigation in Alias and Procedure Lists

Key	Moves to
Uparrow	Previous item on the list
Dnarrow	Next item on the list
Home	First item listed
End	Last item listed
PgUp	Previous page - highlight bar remains on current row
PgDn	Next page - highlight bar remains on current row or last row to contain information
Ctrl-PgUp	First item on the first page
Ctrl-PgDn	Last item on the last page

Information Displays

Windows which are solely for displaying information do not contain a highlight bar. The navigational rules for information displays are detailed in the table below.

Table 8-2 Navigation in Information Displays

Key	Purpose
Uparrow	Scrolls up one line
Dnarrow	Scrolls down one line
PgUp	Scrolls up one page
PgDn	Scrolls down one page
Ctrl-PgUp	Moves to first page
Ctrl-PgDn	Moves to last page

Windows with a double line border allow navigation inside the window through the use of cursor movement keys.

MENU BAR SELECTIONS

Each menu bar selection is outlined below through the use of screen shots and brief descriptions of the pull-down menus.

Control Menu

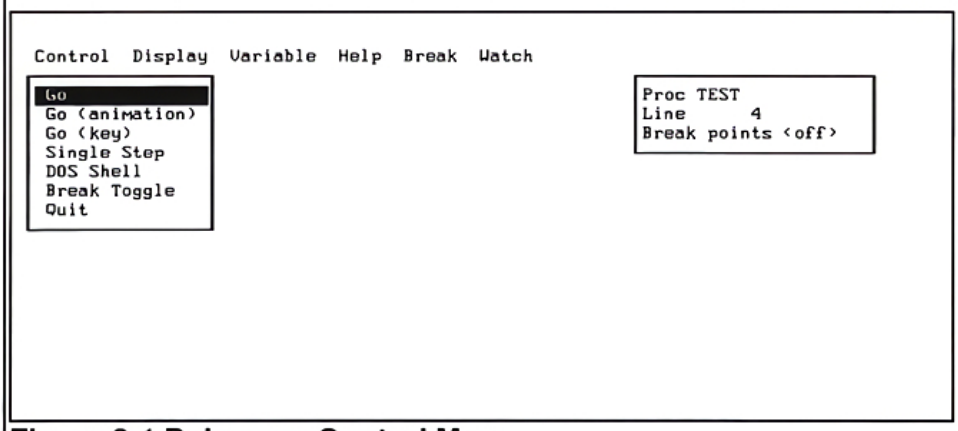


Figure 8-1 Debugger Control Menu

Go: Transfers control to your program, without intervention from the Debugger, until a defined breakpoint or runtime error is encountered or you re-invoke the Debugger by pressing Alt-D.

Go (animation): Displays the Watch Box from line transition to line transition. In all other aspects it operates the same as Go. To slow down the display of the Watch Box (so that it does not have the blinking quality) add

INKEY(expN) as one of the expressions in the Watch menu. For example, INKEY(.1) will cause the display to pause approximately 1/10 of a second per line.

Go (key): Executes your program without intervention from the Debugger after **any key** is pressed. This allows you to stuff a character into the keyboard buffer. In all other aspects it operates the same as Go.

Single Step: Executes your program one line at a time and displays the **next** executable line number (comments and empty lines are ignored). This allows you to check the status of your program as each line is executed.

DOS Shell: Loads another copy of COMMAND.COM. This gives you access to the DOS Command Processor. Any program (restricted only by available memory) can be run. Type "EXIT" at the DOS prompt to return control to the Debugger.

Break Toggle: The breakpoint system is automatically turned on when a breakpoint is defined. This tells the Debugger to ignore all breakpoints. The current status of the breakpoint system is shown in the Watch Box. Individual breakpoints may be toggled on and off from the Break menu.

Quit: Closes all files and returns you to DOS.

Display Menu

```
Control  Display  Variable  Help  Break  Watch

         Expression                          Proc TEST
         Trace                               Line      2
         Status                              Break points <off>
         Database
```

Figure 8-2 Debugger Display Menu

Expression: Expressions can be evaluated and displayed. When a valid expression is entered, the current evaluation for that expression is displayed. To return to the Display Menu, press the Esc key.

Trace: Displays the current procedure name and the names of all programs, procedures and functions which have been executed to reach the current line number of the active procedure. Trace also displays the line number for each procedure showing where the program branched from that procedure to the next level. The most recently activated procedure is listed first.

Status: Displays the current status of most of the SET commands.

Database: Displays two information windows and an active function key list on the screen. The left hand window contains a list of all the active database aliases. Database files are selected by using Uparrow and Dnarrow to highlight the desired database file. If no database files are in USE, the message "WAITING" appears at the bottom of the alias window.

The function key list allows you to use function keys to choose one of four different views of database organization. The selected view remains the default view until another is selected.

F1 - Overview

Gives a general overview of the highlighted database, such as index number, current record number, EOF status, etc.

F2 - Relations

Displays the database files related to the highlighted database file and the current value of the relation expression.

F3 - Indexes

Displays the number and key expression for the highlighted database indexes.

F4 - Structure

Displays the name, type, length, and number of decimals of each field within the highlighted database file. If the structure contains more than 16 fields, a rightarrow activates the right window to allow scrolling. See Table 8-2 for navigational information.

Information corresponding to the chosen view and highlighted database file is displayed in the right hand window of your screen. As you highlight different database files from the active database list and/or choose different database views, the information in the right hand window changes.

Variable Menu

```
Control  Display  Variable  Help  Break  Watch

                 Assign Private
                 Assign Public
                 View Privates
                 View Publics

                                    Proc TEST
                                    Line      2
                                    Break points <off>
```

Figure 8-3 Debugger Variable Menu

Assign Private/ Public: Private or public memory variables for the current procedure can be created or modified with this option. You are prompted for the variable name and new value. When entering public or private variables, the Debugger indicates whether an error or undefined element exists. If either of these conditions is true, the variable is created but is undefined. To return to the Variable Menu, press the Esc key.

View Private: Displays two information windows in the middle of the screen. The left hand window displays a list of active procedures with the current procedure shown at the top. The private variable names, type and contents for the highlighted procedure are shown in the right hand window. Use rightarrow to activate the right window to allow scrolling. To return to the Variable Menu, press the Esc key.

View Public: Displays the public variable names, type and contents. If no public variables have been declared, the message "WAITING" appears at the bottom of the window. To return to the Variable Menu, press the Esc key.

Help Menu

Help is available for each of the menu bar selections. Use the cursor movement keys to move from one selection to another and press Return to confirm your selection.

```
Control  Display  Variable  Help  Break  Watch

                    Control                Proc TEST
                    Display                Line       4
                    Variable               Break points <off>
                    Help
                    Break
                    Watch
                    Speed Keys
                    About
```

Figure 8-4 Debugger Help Menu

The Help Menu provides a list of speed keys which may be used in the Debugger. Speed keys allow you to navigate within the Debugger without the use of arrow keys. A list of the speed keys is shown below:

Table 8-3 Speed Keys

Key	Purpose
Alt-Z	Control menu
Alt-X	Display menu
Alt-V	Variable menu
Alt-W	Watch menu
Alt-P	Set a watch point and return
Alt-H	Help menu
Alt-F	Execute specific help and return
Alt-B	Set breakpoint and return
Alt-S	Single step and return
Alt-G	Run program
Alt-A	Run program with animation
Alt-K	Run program with keystroke
Alt-Q	Quit

Break Menu

The breakpoint system is automatically turned on when a breakpoint is defined. Up to 16 separate breakpoints can be set in any combination of logical expression and procedure line breaks.

```
Control  Display  Variable  Help  Break  Watch

                                        Proc TEST
                                        Line      2
                                        Break points <off>

  Toggle  Line  Expression  Delete
```

Figure 8-5 Debugger Break Menu

Toggle: Individual breakpoints can be turned off. Enter the number of the breakpoint you want to disable and press Return. To disable all breakpoints use the Break Toggle option in the Control Menu.

Line: Enter a line number indicating where you would like a breakpoint to occur. Enter the procedure name and line number. Pressing Return at the Proc prompt automatically selects the current procedure name. If you are not sure where you would like to set a breakpoint for the entered procedure, enter 0 for the line number. The breakpoint system will stop program execution at the first and all subsequent lines of the procedure.

Expression: Define a breakpoint by entering a logical expression. The procedure execution stops when the expression becomes true. Enter the procedure name and the logical expression breakpoint.

When entering breakpoints the Debugger indicates if an expression is not logical, contains an error, or an undefined element. If any of these conditions exist, the breakpoint is not accepted.

If a breakpoint expression becomes undefined during execution, it will not be evaluated. You must delete it manually to remove it from the breakpoint list. The memory variable list will indicate when an element becomes undefined.

Delete: Delete breakpoints by entering the number of the breakpoint you want to delete and press Return.

Watch Menu

The default Watch Box displays the current procedure name, the current line number and the breakpoint system status. You may add up to 16 additional expressions to display and track. The expressions can be of any type. Only the first 20 characters of the evaluated expression are displayed within the Watch Box. If an expression becomes undefined the message "undefined" appears in angle brackets.

```
Control  Display  Variable  Help  Break  Watch

Proc TEST                          Toggle
Line      Z                         Set
Break points <off>                  Move
```

Figure 8-6 Debugger Watch Menu

Toggle: Toggles the Watch Box on and off.

Set: Defines and maintains watch point expressions.

Toggle: Expression watch points can be toggled on and off. Enter the number of the expression watch point you wish to toggle and press Return. Expressions which have been turned off are replaced on the screen with "<off>".

Add: New expression watch points can be added by typing in the expression and pressing Return.

Delete: Delete expressions by typing in the expression number to remove and press Return.

Move: This moves the Watch Box from one side of the screen to the other.

9 The Runtime Environment

Chapter 9 contains information and instructions regarding Clipper and the DOS environment. Topics covered include:

- The DOS command processor
- DOS files and buffers
- ANSI terminal support
- Computer memory usage
- Modifying the runtime environment

The DOS environmental considerations affect compiled and linked programs only. Modifying the environment as described below does NOT affect the operation of the Clipper compiler or the PLINK86-Plus Link Editor.

THE DOS COMMAND PROCESSOR-COMMAND.COM

When you run applications which include external programs (using the RUN command), DOS must be able to locate the DOS command processor (COMMAND.COM). If your program and the DOS command processor do not reside in the same directory, you can determine where DOS looks for the command processor using the DOS "SET" command.

Enter the DOS "SET" command and press RETURN. The screen displays the current status of the DOS environment. The statement that specifies where DOS will look for COMMAND.COM is the "COMSPEC=" statement. To change where DOS looks for the COMMAND.COM program, enter the following DOS command:

C> SET COMSPEC=d:\<path>\COMMAND.COM

where:

"d:" is the disk drive identification letter

"Path" is the directory path to COMMAND.COM.

DOS FILES AND BUFFERS

Database applications require more extensive disk utilization than other types of programs. DOS permits you to alter the environment to optimize your disk utilization.

The DOS "FILES" parameter tells DOS how many files can be open at any given time. The PC-DOS default value is eight, which means DOS will accept eight open files at any given time. A Clipper compiled program can open a maximum of 15 files (255 with DOS 3.3 - See "Open Files" in this chapter). The exact number of files opened depends on your application. You may want to direct DOS to increase the number of files which can be opened when using Clipper.

The DOS "BUFFERS" parameter designates how many buffers DOS can use during a disk read/write operation. The PC-DOS default value is two. After considerable testing, it was found the optimum number of buffers for Clipper is eight.

The FILES and BUFFERS parameters can be changed by creating a CONFIG.SYS file, if the file does not already exist. DOS looks for the CONFIG.SYS file each time the computer is turned on or rebooted. When the file is found, the operating system configures the environment according to the contents of the file.

The CONFIG.SYS file should contain the following two statements:

```
FILES=20
BUFFERS=8
```

It is suggested you use 20 as the number of FILES which can be opened (DOS uses five of these files), and eight as the number of buffers.

Note: DOS 3.3 allows a maximum of 255 open files. Earlier versions allow 20 (five of which are used by DOS).

See your DOS manual for more information about the FILES and BUFFERS parameters.

ANSI TERMINAL SUPPORT

For those computers or applications that require ANSI terminal support, Clipper includes an ANSI terminal support program (ANSI.OBJ).

The ANSI.OBJ file can be linked to your application when you create your executable load module with the Link Editor.

The ANSI terminal support does not define cursor movement or function keys. Consequently, when you use the ANSI terminal support, cursor movement and function keys have no affect.

Using the ANSI.OBJ file requires the presence of the ANSI device driver (ANSI.SYS) on your PC. You must include the ANSI.SYS device driver in your CONFIG.SYS file. To accomplish this, include the following statement in your CONFIG.SYS file:

```
DEVICE=ANSI.SYS
```

When this statement is present in the CONFIG.SYS file, the ANSI device driver will be placed into the DOS environment when the system is turned on or rebooted.

Terminal support programs are also available for Rainbow and WANG PCs and Texas Instrument Professionals. See Chapter 7, *"Compiling And Linking Your Programs"*, for information on linking these object modules to your application.

COMPUTER MEMORY USAGE

Clipper has several features which permit you to control and allocate the available memory in your computer. Clipper also makes use of Expanded Memory.

When a Clipper compiled program runs, it allocates and uses the available memory for several different purposes:

- Memory variables
- Buffers used to perform indexing
- Execution of other programs from within your application (RUN command)
- A "free memory pool" for data manipulation

If you do not wish to specifically allocate the memory usage, the Clipper compiled program automatically distributes the available memory for the above purposes. The Clipper program checks, however, to see if you have used the DOS "SET" command to instruct it to alter memory allocation.

The automatic memory allocation provided by the Clipper program performs very well for most applications. There are some instances, however, when you will wish to specifically control how your application allocates memory.

Usual reasons for overriding the automatic memory allocation are to shift memory to the category(s) where an application needs it most or to change the application's performance characteristics.

Automatic Memory Allocation

If you do not use a DOS "SET" command to alter the memory allocation, the Clipper program will automatically allocate the available memory in the following sequence.

1. The executable load module is loaded into memory above DOS.

2. 24K bytes are reserved for a free memory pool.

3. Of the remaining available memory, 20 percent is allocated for memory variables (44K maximum).

4. Of the memory remaining after step 3, 33 percent is allocated for RUN commands and index buffers (16K minimum).

5. The remaining available memory is added to the previously reserved 24K bytes, and allocated to the "free memory pool" to be used for program processing. Specific uses of "free memory pool" include storage of character variables, arrays, and transient copies of database records from open .dbf files.

Accordingly, the absolute minimum memory required above DOS to run your application is the load module size, 24K for the free pool, 16K for index buffers, plus the allocation for memory variables.

If Expanded Memory is present, the Clipper program will automatically make use of it. The Clipper program will allocate up to one megabyte, if available, for index buffers. The allocation described in step 4 above will only be used for RUN commands. If Expanded Memory is present, the Clipper program will require a minimum of 16K bytes of the Expanded Memory.

MODIFYING THE RUNTIME ENVIRONMENT

Clipper provides the ability to control allocation of memory by using a DOS "SET" command to modify the DOS environment.

The syntax of the DOS command is as follows:

SET CLIPPER= [Vnnn][;Rnnn][;Ennn][;Xnnn][;Fnnn]

In each of the above parameters, "nnn" is the number of K bytes. For example, if you entered the number "128" for one of the parameters, the number used will be 128K bytes.

> **Note:** A space between "CLIPPER" and the "=" will cause the memory directive to be ignored by your application.

You can enter the "SET CLIPPER=" command at the DOS prompt, or you can include it in your AUTOEXEC.BAT file. At load time, Clipper compiled programs look for any values entered using the above SET command and then allocate the memory in accordance with the values established for each of the parameters.

Memory Variables

When you enter a value for the "V" parameter, the Clipper program restricts the amount of memory allocated for a memory variable table to the amount specified. If this parameter is not specified, Clipper automatically allocates 20 percent, up to a maximum of 44K bytes, of memory for the memory variables in your application.

Clipper permits a maximum of 2048 memory variables. Each variable requires 22 bytes of memory for its name, type, and value (or memory location if a character type).

If your program uses fewer memory variables than the maximum permissible, you can limit the amount of memory space allocated for this purpose. For example, if your program uses a maximum of 256 memory variables, you could enter "V006;" for this parameter. The calculation is as follows:

256 memory variables x 22 bytes = 5,652 bytes

Divide the number of bytes by 1024 (1 K bytes) and round the resulting number to the next higher K (1024) bytes.

5,652 / 1024 = 5.5K bytes (6K)

> **Caution:** If you enter a number for the "V" parameter larger than 44K, the Clipper program will indeed allocate memory in accordance with the number you enter. However, any amount in excess of 44K bytes is unavailable for any processing by the Clipper program and thus the excess is allocated for no usable purpose.

The presence or absence of Expanded Memory has no affect on this parameter. Memory variables are not placed in Expanded Memory.

Buffers and the RUN Command

The number entered for the "R" parameter reserves an area in memory for both buffers (used for indexing) and external programs executed using the RUN command.

> **Note:** The use of the word "buffers" in this section is NOT the same as, nor does it have any affect on, the DOS "BUFFERS=" value placed in the CONFIG.SYS file.

Space allocated for RUN commands and buffers is shared. When your application executes a RUN command, the external program being executed uses as much of the memory space as is required. Space currently being occupied by buffers is relinquished for the external program as it is being executed.

If you enter a number for this parameter which is too large, larger than the available memory left in the computer, an error will occur ("System error - not enough memory").

If you have Expanded Memory, memory use for the "R" parameter will change. The amount allocated for "R" will be used only for RUN commands. Index buffers will be moved to Expanded Memory unless you specify "E000;" (see the "E" parameter below).

Expanded Memory

Use the "E" parameter to identify the maximum amount of Expanded Memory you wish the Clipper compiled program to use. If you do not enter the "E" parameter, and an Expanded Memory Board is installed, the Clipper compiled program allocates all of the available Expanded Memory, up to one megabyte.

The Clipper application uses Expanded Memory for index buffers only. It is not used for other purposes. Indexing operations are usually speeded up by using Expanded Memory.

If you enter this parameter, the minimum number allowable is 16K bytes ("E016;").

Excluding Memory

The "X" parameter excludes the specified amount of memory from being allocated and used by the Clipper application.

This parameter is provided primarily for testing purposes. When you use this parameter and execute an application, the program allocates the memory as if the amount of memory specified is not available. For example, if you enter "X128;" for this parameter, and your computer contains 640K of memory, when you execute a Clipper program, the program will allocate memory as though the computer contains only 512K of memory.

There is an exception to this. External programs using the RUN command can use available memory excluded with the "X" parameter.

"X" parameter memory is excluded immediately after the application loads, before it allocates memory to other purposes. The resultant amounts allocated to these purposes are diminished accordingly (unless separately controlled by the "V" and/or "R" parameters.)

Open Files

The "F" parameter indicates to Clipper the maximum number of files you expect to have open at any one time (the default in DOS is eight). This parameter is used in conjunction with the CONFIG.SYS "FILES" parameter. Clipper uses the smaller of the two parameters to determine the number of files which may be opened.

> **Note:** Only DOS 3.3 will allow more than 20 files to be opened by a single process. The maximum FILES supported by DOS 3.3 is 255.

Examples:

CONFIG.SYS **files = 120**
buffers = 8

SET CLIPPER = F50

In this case, Clipper will allow 50 files to be opened (five used by DOS).

or

CONFIG.SYS **files = 20**
buffers = 8

SET CLIPPER = F50

Clipper will allow 20 files to be opened (five used by DOS).

The "F" parameter should be set judiciously. It must be large enough to allow the use of all your needed files and yet not so large that you unnecessarily take up valuable memory space.

10 Using Clipper With A Local Area Network

Chapter 10 contains information and instructions regarding the use of Clipper in a Local Area Network environment.

Topics covered include:

- What is a Local Area Network?
- Clipper features.
- Network commands.
- Network functions.
- Compatibility of Local Area Networks with Clipper.
- Programming for a Local Area Network environment.
- Effects of the network environment on files.
- Effects of the network environment on commands.
- Locks.prg source code.

WHAT IS A LOCAL AREA NETWORK?

A Local Area Network (LAN) allows two or more personal computers to share data and peripheral equipment in a controlled environment.

Connecting the personal computers requires additional equipment and software. Network software must be installed to control the computers. Usually, the required equipment consists of a board installed inside each of the computers and a cable which connects those boards. In a simple network, one main computer (the file server) usually contains the File Server program, the hard disk containing the files to be shared, and the attached peripherals. One or more computers are attached to the network in order to share the files and the peripherals. The network software identifies the shared equipment and controls which computer can use the shared equipment and data.

Sharing data is possible because of a method for opening files (called shared mode) provided by the network operating system (and utilized by Clipper) that lets two or more users open the same file. The control comes in the form of functions that temporarily override the sharability of data in favor of private control by a single user.

CLIPPER FEATURES

Clipper provides the following features which permit you to develop applications for use on a Local Area Network:

- The ability to open files either on a shared or an exclusive basis (SET EXCLUSIVE/USE...EXCLUSIVE commands).
- Logical file locking to prevent two or more users from updating the same file at the same time (FLOCK() function).
- Logical record locking to prevent two or more users from updating the same record in a file at the same time (RLOCK() function).
- Shared mode to allow two or more users to access the same record simultaneously (SET EXCLUSIVE OFF).
- A command to redirect printer output (SET PRINTER TO command).
- The ability to test the current lock status of a file or record (FLOCK() and RLOCK()).

- The ability to release previous file and record locks (UNLOCK [ALL] command).

- The ability to test whether USE, USE...EXCLUSIVE or APPEND BLANK were used successfully (NETERR() function).

The network commands and functions used to design networking applications are listed below. See Chapters 5 and 6 for detailed descriptions and examples.

NETWORK COMMANDS

SET EXCLUSIVE ON/off

Determines whether database and index files are opened on a shared (off) or exclusive (ON) basis.

SET PRINTER TO <destination>

Establishes the destination of the printed output.

UNLOCK [ALL]

Releases file and record locks within the current work area previously set by the lock functions.

USE [<filename>] [INDEX <index file list>] [EXCLUSIVE] [ALIAS <alias name>]

Opens a database and its associated index file for exclusive use.

NETWORK FUNCTIONS

FLOCK()

Attempts to logically lock the file and returns true (.T.) if the attempt was successful. FLOCK() releases any record or file locks previously placed on the file by the same user.

NETERR()

Returns true (.T.) if a USE, USE...EXCLUSIVE or APPEND BLANK command fails in a network environment. These commands fail in the following circumstances:

Table 10-1 Causes of NETERR() returning true

Command	Cause of Failure
USE	USE...EXCLUSIVE by another process
USE...EXCLUSIVE	USE or USE...EXCLUSIVE by another process
APPEND BLANK	FLOCK() by another process or two APPEND BLANK attempts at the same time

RLOCK()/LOCK()

Attempts to logically lock the record and returns true (.T.) if the attempt was successful. RLOCK() releases previous FLOCK()s placed on the database file, or RLOCK()s placed on any record within the database file by the same user.

The Clipper locking system offers extensive flexibility and optimal computer usage. It should not be assumed, however, that Clipper locks and those of other network software products, such as dBASE III PLUS, will be compatible.

COMPATIBILITY OF LOCAL AREA NETWORKS WITH CLIPPER

Clipper's compatibility with Local Area Networks relies on adherence to DOS 3.1 or greater function calls. Clipper uses DOS calls exclusively for all network related operations. Consequently, Clipper-compiled applications are compatible with network products designed to the DOS standard.

You should be familiar with the nature and requirements of your Local Area Network products before attempting to develop applications that use the Clipper network features.

> **Note:** You must be running under PC/ MS-DOS version 3.1 or greater on any network work-station using files in a shared mode.

PROGRAMMING FOR A LOCAL AREA NETWORK ENVIRONMENT

To write effective network applications, you should be familiar with these key issues:

- When to lock records and files.
- How to open files for sharing.
- The mechanics of locking.
- How to attempt a lock: the issue of persistency.
- How to resolve a failed lock: the issue of response.

> **Note:** If your application does NOT require access to a shared database file, even though your computer system has a Local Area Network, the information on the following pages does not apply to your application. A Clipper application, by default, assumes a single user and a non-shared database file (SET EXCLUSIVE ON).

When to Lock Records and Files

Once you determine that you need to share the information in a database file, you must determine when file or record access by more than one user must be prevented. You must make these determinations as the application is developed in order to protect the integrity of your data.

Identify any process in your application which, when executed, will require a single user to have control and the EXCLUSIVE USE of a database file. In Clipper, this is reduced to two basic guidelines:

- Locking is REQUIRED whenever you are going to write to a database file.
- Locking is OPTIONAL at all other times.

The commands that write to database files are REPLACE, DELETE, RECALL, and @...GET <fieldname>. Pre-locking is a hard and fast requirement for these commands. Clipper enforces it with the error message "System error not locked" if you execute them without a record lock when SET EXCLUSIVE is OFF.

Files should be locked or USEd EXCLUSIVEly for processes that update multiple records in a database file, such as APPEND FROM, DELETE, and UPDATE ON. You do not want another user updating the file in the middle of the process.

You should also lock a file when performing a process that reads information from each record in the database file, such as COUNT, SUM, and TOTAL, even though locking is not required by the software. These processes do not update the file. However, you want to make sure the data, and therefore the results, do not change during the process.

Certain operations (PACK, REINDEX and ZAP) cannot be performed in the shared mode; they must have EXCLUSIVE USE of the file. Clipper enforces this requirement with the error message "System error not exclusive" if you attempt to use these commands without EXCLUSIVE USE of the database file.

Clipper does not require a lock for operations that only read files, such as LIST, SEEK, REPORT FORM, and GO TO. Unlike other related database products, you may APPEND BLANK records to a shared database file with Clipper.

Note: When a Clipper application is executed, file access to the database and its associated files will automatically be set to "non-shared." Network operation will require you to include the command "SET EXCLUSIVE OFF" in your application in order to share the information in database files.

How to Open Files for Sharing

There are a few simple rules for shared file usage.

- Open files in the shared mode by SETting EXCLUSIVE OFF before issuing the USE command.
- Always check NETERR() immediately after a USE to make sure the file was successfully opened.
- Never include the INDEX clause in a USE command. Instead, defer opening the index files until after you check NETERR(), and do it with SET INDEX.

The following code illustrates these rules:

```
SET EXCLUSIVE OFF   && Later USE will be shared mode
USE File
IF NETERR()         && NETERR() tells you if
*                                you USEd OK
   ? 'File not available in shared mode. ' +;
   'Program terminated.'
   QUIT
ENDIF
SET INDEX TO...
```

The following code is very similar to that shown above, but uses the recommended network file USE function found in the Locks.prg procedure file:

```
SET EXCLUSIVE OFF
IF Net_use("File",.F.,5)
   SET INDEX TO...
ELSE
   ? 'File not available in shared mode.'+;
   'Program terminated.'
   QUIT
ENDIF
```

Notice that you must anticipate the file's possible unavailability (even in the more permissive shared mode) since another user could have the file open in the exclusive mode. That exclusiveness means you are sometimes excluded.

If this is the only application on the network that ever uses the file, of course, that will never happen (since you only use it non-exclusively). But you should not rely on that assumption. If a curious programmer uses the file with DOT.PRG, your program

should be able to detect it. It is better to take the safe and conservative approach.

You should always separate SET INDEX from USE. A failed open attempt does not disrupt the program if it is a database file, but does if it is an index. A failed USE <database> merely sets NETERR() true (.T.), but a USE <database> INDEX <index> that is denied shared use of the index causes a run-time error.

The problem is solved by recognizing that if the shared-mode USE <database> succeeds, opening the index successfully is guaranteed, since an index file's open mode is dictated by the associated database file. If the shared mode USE <database> works, any other users of the database running this application must also be in shared mode. That mode extends to their use of the index files as well. The shared mode attempt to open index files with SET INDEX TO will therefore succeed.

The Mechanics of Locking

Exclusive control of shared data results from use of the lock functions, FLOCK() and RLOCK(). The lock functions both perform the locking attempt and report back the result of the attempt (true or false). If User 1 executes a successful lock function the effects are as follows:

In the case of an FLOCK(), the entire file in the current work area is locked. In the case of an RLOCK(), only the record that is current to User 1 at the time of the RLOCK() is locked.

User 1 is now allowed to write. (Writing to unlocked shared files normally returns the "System error not locked" message.) Other users' attempts to lock the same record or file will fail. There are no effects on other users' read access attempts.

These effects last until User 1 releases the lock by:

- Issuing an UNLOCK in the locked file's select area.
- Issuing UNLOCK ALL in any select area.
- Closing the file with USE.
- Terminating the program normally.
- Issuing another lock command in that file.

> **Note:** Attempting a lock undoes an existing lock before trying to achieve the new one. Any failed lock, or a successful RLOCK() on a different record than had been locked, lifts the lock that had been in place.

One lock per work area can be in effect at a time in any file held open by a user in shared mode.

Attempting a Lock: The Issue of Persistency

When laying strategy for getting the lock, an issue of "persistency" arises. If you cannot lock what you want, when is giving up premature, and when is it appropriate? Several approaches are possible:

- Try once.
- Retry fixed time.
- Retry until interrupted (Esc).
- Retry fixed time or until interrupted.
- Retry forever.

The "try once" approach is not persistent enough; the "retry forever" approach is too persistent. Both are impractical for most applications. To make your network programs both flexible and practical, you need a compromise. There are several alternatives for deciding when to terminate the retry effort: set a time limit, let the user terminate it, or use a combination of the two.

The user-defined functions REC_LOCK() and FIL_LOCK() in the Locks.prg file provide for the time limit. A slight modification permits user termination. Use the functions of Locks.prg as a starting point and adapt them to suit your applicaton.

Resolving a Failed Lock: The Issue of Response

If you find that your target data is unavailable (RLOCK() and FLOCK() remain false (.F.)), it is time for contingency plans. Here lies the real difference between network code and familiar single user code.

The code adaptations for opening files and attempting locks are straightforward. They do not change the program's flow or function, but do add a few more activities along the way. If, however, needed records or files are locked, the program has to abandon its original intentions and come up with substitute plans.

As the programmer you should do two things at this point:

- Communicate the lock failure to the user.
- Make a branching choice.

You can make the communication transient (two seconds' duration, for example), or follow it by an INKEY(0) pause to be sure the user has seen it. It can be a small message in a corner of the screen or a whole screen sandwiched between SAVE SCREEN/RESTORE SCREEN commands.

The branching can be hard-coded or can provide for multiple-way branching with user intervention at a small menu. In that case the menu itself constitutes the user advice. A typical menu might offer three choices:

- Retry.
- Try same activity, different data.
- Abort current activity - go back to the menu.

The first choice allows the user to extend the lock effort. It does not genuinely seek to resolve the lock's failure.

The second choice makes sense in many applications. If the target record is not available, use another instead. Come back to the original later and it will probably be free. For example, a data entry operator working on a stack of credit adjustment slips will not mind putting Smith's slip at the bottom of the pile and moving on to Jones and Brown. When he gets back to Smith an hour later, that record will be free. Therefore, make the code branch to the record selection entry point.

The third choice implies abandoning not only the record in question, but the activity itself. The user leaves the credit adjustment section of the program altogether and goes back to the main menu.

Whatever choice you make, be sure to UNLOCK as soon as possible after your lock has served its purpose. If your response involves a menu, UNLOCK before you display it so that you do not keep files locked needlessly.

EFFECTS OF THE NETWORK ENVIRONMENT ON FILES

When you enable the network (or shared) environment by executing the SET EXCLUSIVE OFF command, files behave differently than in an exclusive (or non-shared) environment. Databases and their associated files are opened for shared or exclusive use depending on the current status of the SET EXCLUSIVE command and/or whether you have opened the file with the USE or the USE...EXCLUSIVE command.

If you SET EXCLUSIVE ON, a database and its associated files subsequently opened with the USE command are opened on an exclusive (non-shared) basis. If you SET EXCLUSIVE OFF and then open the file with a USE command, database files are opened on a shared basis.

If you SET EXCLUSIVE OFF and subsequently open a database file with the USE...EXCLUSIVE command, the file is opened on an exclusive (non-shared) basis.

> **Note:** File and record locking are needed only when you have opened database files on a shared basis.

Some file handling commands open a database (and/or the related .dbt and index) file other than the file currently in USE. Upon completion of the command execution, the file is closed.

The following table shows how the file not currently in use will be opened during the execution of the command.

Table10-2 How Files are Opened at Command Execution

Commands That Read From <file> and Open <file> Shared	Commands That Write To <file> and Open <file> Exclusive
APPEND FROM <file>	COPY STRUCTURE TO <file>
CREATE...FROM <file>	COPY TO <file>
LABEL FORM <file>	CREATE <file>
REPORT FORM <file>	INDEX ON...TO <file>
RESTORE FROM <file>	JOIN...TO <file> ...
TYPE <file>	SAVE TO <file> ...
UPDATE...FROM <file>...	SET ALTERNATE TO <file>
	SORT...TO <file>
	TOTAL...TO <file>

A file opened for shared use, as shown above, is opened using the DOS read-only attribute.

The SET INDEX TO command will open the index file for shared or exclusive use depending on the current status of the associated database file.

Some of these commands work with two files, often the one named in the above syntax and the current database file. The table refers only to the named file. The current database file's open-mode is predetermined, under your explicit control, and depends on commands previously discussed.

EFFECTS OF THE NETWORK ENVIRONMENT ON COMMANDS

Some commands require exclusive use of the file, or locked files or records, to function properly in the network environment. Notice that the requirements will change depending on whether the command is operating on a single record or multiple records. The requirements of these commands are indicated below.

Table 10-3 Effects Of The Network Environment On Commands

Command		Requirement
@...SAY...GET		RLOCK()
APPEND FROM		USE...EXCLUSIVE or FLOCK()
DELETE	(single record)	RLOCK()
DELETE	(mutiple records)	USE...EXCLUSIVE or FLOCK()
PACK		USE...EXCLUSIVE
RECALL	(single record)	RLOCK()
RECALL	(multiple records)	USE...EXCLUSIVE or FLOCK()
REINDEX		USE...EXCLUSIVE
REPLACE	(single record)	RLOCK()
REPLACE	(multiple records)	USE...EXCLUSIVE or FLOCK()
UPDATE ON		USE...EXCLUSIVE or FLOCK()
ZAP		USE...EXCLUSIVE

LOCKS.PRG SOURCE CODE

Locks.prg, located on your Clipper disk, contains user-defined functions for use in your networking application. You should use the functions ADD_REC(), FIL_LOCK(), NET_USE() and REC_LOCK() instead of APPEND BLANK, FLOCK(), USE...EXCLUSIVE and RLOCK() respectively. Each of these functions attempts to lock the record or file at defined intervals for a specified length of time so that computer time is optimally used. To use these functions, include the "SET PROCEDURE TO Locks" command in your program. Locks.prg assumes that SET EXCLUSIVE is OFF. The Locks.prg source code is listed below.

Net_use Function

The NET_USE() function tries to open a file for exclusive or shared use and passes the following parameters:

Character -	name of the database file to open
Logical -	mode of open (exclusive/.NOT. exclusive)
Numeric -	seconds to wait (0 = wait forever)

If the Net_use is successful, then SET INDEXes in the calling procedure.

```
IF NET_USE("Accounts", .T., 5)
   SET INDEX TO Name
ELSE
   ? "Account file not available"
ENDIF

FUNCTION Net_use
PARAMETERS file, ex_use, wait
PRIVATE forever
forever = (wait = 0)
DO WHILE (forever .OR. wait > 0)
   IF Ex_use            && Exclusive
      USE &file EXCLUSIVE

   ELSE
      USE &file         && Shared
   ENDIF
   IF .NOT. NETERR()    && USE succeeds
      RETURN (.T.)
   ENDIF
   INKEY(1)             && Wait 1 second
   wait = wait - 1
ENDDO
RETURN (.F.)            && USE fails
* EOF Net_use
```

Fil_lock Function

The FIL_LOCK() function attempts to lock the current shared file. A numeric parameter is passed indicating the number of seconds Fil_lock will continue attempting a lock before returning false for an unsuccessful lock.

```
IF FIL_LOCK(5)
   REPLACE ALL Price WITH Price * 1.1
ELSE
   ? "File not available"
ENDIF

FUNCTION Fil_lock
PARAMETERS wait
PRIVATE forever
IF FLOCK()
   RETURN (.T.)                && Locked
ENDIF
forever = (wait = 0)
DO WHILE (forever .OR. wait > 0)
   INKEY(.5)                   && Wait 1/2 second
   wait = wait - .5
   IF FLOCK()
      RETURN (.T.)             && Locked
   ENDIF
ENDDO
RETURN (.F.)                   && Not locked
* EOF Fil_lock
```

Rec_lock Function

REC_LOCK() attempts to lock the current shared record. A numeric parameter is passed indicating the number of seconds REC_LOCK() will continue attempting a lock before returning false for an unsuccessful lock.

```
IF REC_LOCK(5)
   REPLACE Price WITH newprice
ELSE
   ? "Record not available"
ENDIF

FUNCTION Rec_lock
PARAMETERS wait
PRIVATE forever
IF RLOCK()
   RETURN (.T.)                && Locked
ENDIF forever = (wait = 0)
DO WHILE (forever .OR. wait > 0)
   IF RLOCK()
      RETURN (.T.)             && Locked
   ENDIF
   INKEY(.5)                   && Wait 1/2 second
   wait = wait - .5
ENDDO
RETURN (.F.) && not locked
* EOF Rec_lock
```

Add_rec Function

The ADD_REC() function attempts to add a new locked record to the database. A numeric parameter is passed indicating the number of seconds to continue attempting an append before returning false for an unsuccessful append.

```
FUNCTION ADD_REC
PARAMETERS wait
PRIVATE forever
APPEND BLANK
IF .NOT. NETERR()
   RETURN (.T.)
ENDIF forever = (wait = 0)
DO WHILE (forever .OR. wait > 0)
   APPEND BLANK
   IF .NOT. NETERR()
      RETURN .T.
   ENDIF
   INKEY(.5)                  && Wait 1/2 second
   wait = wait - .5
ENDDO
RETURN (.F.)                  && Not locked
* EOF Add_rec
```

11 The Extend System

Chapter 11 contains information and instructions on writing user-defined functions in other languages such as C and Assembler. Topics covered include:

- Description of the Clipper Extend System
- Summary of Extend Functions
- Interfacing with C
- Sample C functions
- Compiling and linking C code
- Extend System C interface functions
- Interfacing with Assembly Language
- Sample Assembly routines
- Assembly language Extend macros
- Extend System Assembly language functions
- Summary of Changes from Autumn '86

DESCRIPTION OF THE CLIPPER EXTEND SYSTEM

Clipper allows you to write functions in other languages and provides tools to pass and return parameters to your functions. Your functions may then be used like any other function in the Clipper language. For example, you may assign the value returned to a memory variable or display it on the screen.

```
memvar = MyFunc(<exp>)
?  MyFunc(<exp>)
```

You may write Extend System functions in C, or 8086 assembler and link them with your compiled application. The functions in C and Assembler can be used to get information to which the Clipper language does not have access, such as database file structures or low level information available only through DOS interrupts.

When you create a new user-defined function, you must take care in choosing a name that does not conflict with internal functions or their four letter or longer abbreviations. Your newly named function must be used in your code exactly as declared. No abbreviations are permitted at all, so name your user-defined functions exactly as you want to use them.

User-defined functions in C and Assembler, must be used explicitly in the application or be declared EXTERNAL at compile time in order for them to be included as symbols at link time. User-defined functions in REPORT or LABEL FORMs or in INDEX key expressions must be declared EXTERNAL. For example:

```
EXTERNAL MyFunc, YourFunc
```

SUMMARY OF EXTEND FUNCTIONS

Table 11-1 Summary of Extend Functions

Clipper Type	C Prototype		ASM
character	char*	_parc(int [,int])	__PARC
date	char*	_pards(int [,int])	__PARDS
logical	int	_parl(int [,int])	__PARL
numeric	int	_parni(int [,int])	__PARNI
numeric	long	_parnl(int [,int])	__PARNL
numeric	double	_parnd(int [,int])	__PARND
	int	_parclen(int [,int])	__PARCLEN
	int	_parcsiz(int [,int])	__PARCSIZ
	int	_parinfa(int, int)	__PARINFA
	int	_parinfo(int)	__PARINFO
character	void	_retc(char*)	__RETC
date	void	_retds(char*)	__RETDS
logical	void	_retl(int)	__RETL
numeric	void	_retni(int)	__RETNI
numeric	void	_retnl(long)	__RETNL
numeric	void	_retnd(double)	__RETND
(no value)	void	_ret(void)	__RET
	void	_retclen(char*, int)	__RETCLEN

INTERFACING WITH C

Clipper supports user-defined functions written in C through the Extend System. This is a somewhat higher level interface system than the usual CALLs in that you use specialized functions for passing parameters to and from the routine. These functions reside in the Clipper library. Your C routine must declare them by including EXTEND.H. This file contains the program declarations and macro definitions necessary to write user-defined functions in C.

Receiving Parameters in C

The concept of C formal parameters disappears at this point. All parameters may be accessed using the _par() functions. Therefore, all C functions accessed through the Extend System are declared with no formal parameters.

Returning Values From C

The _ret() functions are used to return values back to Clipper. Each function, by definition, can return only one value. The

choice of which _ret() function you use depends on the data type being returned.

Sample C Function

Once you compute the value you want to return, pass it to the appropriate _ret() function and it becomes the function return value. It is still, however, necessary to execute a C return. _ret() functions do not pass control back to Clipper.

Here is a shell of a typical Clipper user-defined function written in C:

```
CLIPPER <function_Name> ()
/* formal C parameters omitted */
{
   declarations of local variables

   if (<parameters are valid>)
   {
      <code  to execute >
   }
   else
   {
      <code to execute for undefined parameters >
   }
 }
```

In Clipper:

```
DECLARE arr[3]
arr[1]    = "Devorah"
arr[2]    = 456
arr[3]    = CTOD("09/01/87")
Arrfunc(arr)
```

In C:

```
#include "extend.h"     /* Declare Extend System */
#include "conio.h"      /* standard io library */

CLIPPER arrfunc()       /* specify CLIPPER */
                        /* macro to define function */
{
   int x;

   for (x = 1; x  <= _parinfa(1, 0); ++x)
   {
      /* string variables */
      if (_parinfa(1, x) == CHARACTER)
```

```
        {
            cprintf("%s\n", _parc(1, x));
        }

        /* integer or floating point */
        if (_parinfa(1,x) == NUMERIC)
        {
            cprintf("%u\n", _parni(1, x));
        }
        else
            /* dates */
            if (_parinfa(1, x) == DATE)
            {
                cprintf("%s\n", _pards(1, x));
            }
    }
    _ret();
}
```

Note: The use of cprintf() above is for demonstration purposes only. We recommend that all terminal I/O be done through Clipper.

COMPILING AND LINKING C CODE

To compile a Microsoft C 5.0 routine that can be linked with Clipper Summer '87:

C>CL /c /AL /Zl /Oalt /FPa /Gs <filename>.c

where:

CL = the compiler command
/c = (c)ompile without linking
/AL = set program configuration for (L)arge model
/Zl = remove default (l)ibrary-search records from object file
/Oalt = control optimization
where:
a = relax alias checking
l = enable loop optimization
t = favor execution speed

/FPa = floating point (a)lternate library
/Gs = remove calls to (s)tack-checking routine

Predefined C Macros

EXTEND.H sets up several "parameter check" macros that can be used for parameter checking. The following is a table of manifest constants and macros:

Table 11-2 Manifest constants in Extend.h

Type		Returns
undefined	=	UNDEF
character	=	CHARACTER
numeric	=	NUMERIC
logical	=	LOGICAL
date	=	DATE
by reference	=	MPTR /* or'ed with type when passed by reference */
memo	=	MEMO
array	=	ARRAY

Predefined C Interface Macros

Table 11-3 C Interface Macros

EXTEND.H Macro	Defined as
PCOUNT	(_parinfo(0))
ISCHAR(order)	(_parinfo(order) & CHARACTER)
ISNUM(order)	(_parinfo(order) & NUMERIC)
ISLOG(order)	(_parinfo(order) & LOGICAL)
ISDATE(order)	(_parinfo(order) & DATE)
ISMEMO(order)	(_parinfo(order) & MEMO)
ISBYREF(order)	(_parinfo(order) & MPTR)
ISARRAY(order)	(_parinfo(order) & ARRAY)
ALENGTH(order)	(_parinfa(order, 0))

Within the user-defined function, you can use the PCOUNT macro and any of the IS-type macros to determine how many parameters were passed and their type. This enables you to validate parameters ensuring the correct number and type passed. It additionally allows you to pass variable numbers of parameters of various types.

EXTEND SYSTEM C INTERFACE FUNCTIONS

_exmback()

Free allocated memory

Syntax Usage:

```
#include        <extend.h>

void            _exmback(pointer, bytes)

unsigned char   *pointer;       Pointer from _exmgrab()
unsigned int    bytes;          Size passed to _exmgrab()
```

Description:

_exmback() releases the memory allocated by _exmgrab(). Note that the same pointer and size used in _exmgrab must be passed as parameters.

Example:

```
size = 512
buff = _exmgrab(size)       /* allocate memory */
 if (buff)                  /* if successful (!null) */
    check = TRUE;
.
.
.
if (check)
    _exmback(buff, size);   /* deallocate memory */
```

_exmgrab()

Allocate memory

Syntax Usage:

```
#include        <extend.h>

unsigned char   *_exmgrab(bytes)

unsigned int        bytes;      Requested amount of memory
```

Description:

_exmgrab() allocates an amount of memory in bytes. If successful, it returns a char pointer to the allocated space in memory; otherwise, it returns NULL. To free this memory, use _exmback().

Example:

```
size = 512
buff = _exmgrab(size)   /* allocate memory */
 if (buff)              /* if successful (!null) */
    check = TRUE;
```

_parc()

Passes pointer to Clipper character string

Syntax Usage:

```
#include        <extend.h>

char            * _parc(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parc() is used for getting a character parameter from Clipper. Note that _parc() does not make a copy of the passed parameter.

_parclen()

Length of character string

Syntax Usage:

```
#include        <extend.h>

int             _parclen(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parclen() returns the length of a given string. It is used primarily with respect to CHR(0). This has two uses:

- Check for the length of a string with embedded CHR(0)s in it.
- Get the length of a string without counting the null terminator.

_parcsiz()

Allocated size of a character string passed by reference using the "@" symbol

Syntax Usage:

```
#include      <extend.h>

int           _parcsiz(order [, index])

int           order;     Placement in actual parameter list
int           index;     Array element index
```

Description:

_parcsiz() returns the number of bytes in memory allocated for the specified string including the null terminator. Note that _parsize() returns zero for constants, e.g., DO <process> WITH "ABC".

Note also that assigning a value that exceeds this size will write over other memory area.

Example: The following demonstrates the use of the _parcsiz() function.

In Clipper:

```
X = SPACE(40)
X = "DAVE"
UDF(X)
?
UDF(@X)
```

In C:

```
#include <stdio.h>
#include "extend.h"
CLIPPER udf()
{

   printf("Maximum string size %u\r\n", _parcsiz(1));
   printf("Current string length %u\r\n", _parclen(1));

}
```

_pards()

Passes character pointer to Clipper date

Syntax Usage:

#include	<extend.h>	
char	*_pards(order [, index])	
int	order;	Placement in actual parameter list
int	index;	Array element index

Description:

_pards() gets a date parameter from Clipper and returns a character pointer in the form YYYYMMDD. Note that there is only one _pards() pointer on the stack. Therefore you must strcpy() the result to a variable. You cannot simply pass the pointer.

Example:

In Clipper:

```
CLEAR
X = DATE()
Y = CTOD("06/19/56")
Z = CTOD("03/02/52")
UDF(X, Y, Z)
```

In C:

```
#include "\include\extend.h"      /* Declare Extend
                                        System */

CLIPPER udf()
{
char   *ptr1;
char   *ptr2;
char   *ptr3;
char   str1[9];
char   str2[9];
char   str3[9];

   ptr1 = _pards(1);
   ptr2 =_ pards(2);
   ptr3 = _pards(3);

   strcpy (str1, _pards(1));
   strcpy (str2, _pards(2));
   strcpy (str3, _pards(3));

   printf("DATE %s\r\n", _pards(1));

   printf("DATE %s\r\n", _pards(2));

   printf("DATE %s\r\n", _pards(3));

   printf("DATE %s %s %s\r\n", _pards(1), _pards(2),
   _pards(3));

   printf("DATE %s %s %s\r\n", ptr1, ptr2, ptr3);

   printf("DATE %s %s %s\r\n", str1, str2, str3);

}
```

Output:

```
DATE 19880115
DATE 19560619
DATE 19520302
DATE 19880115    19880115   19880115        *Note
DATE 19880115    19880115   19880115        *Note
DATE 19880115    19560619   19520302
```

_parinfa()

Parameter type-checking of array elements

Syntax Usage:

#include	<extend.h>	
int	_parinfa(order [, index])	
int	order;	Placement of the array in parameter list
int	index;	Array element index

Description:

_parinfa() returns type of an array element and is used for parameter type-checking. Since Clipper arrays can have mixed type elements, each one must be type-checked before it can be used.

Note that _parinfa(<order>, 0) returns the number of array elements.

Example:

The following example, ArrFunc(), takes an array defined in Clipper and displays all the elements formatted according to data type to the console.

In Clipper:

```
DECLARE array[2]
array[1] = "Devorah"
array[2] = 456
ArrFunc(array)
```

In C:

```
for (x = 1; x   <= _parinfa(1, 0); ++x)

{   /* string variables */
    if (_parinfa(1, x) == CHARACTER)
    {
        cprintf("%s\n", _parc(1, x));
    }

    /* integer or floating point */
    if (_parinfa(1, x) == NUMERIC)
    {
```

```
        cprintf("%u\n", _parni(1, x));
    }
}
```

_parinfo()

Parameter type checking

Syntax Usage:

```
#include        <extend.h>

int             _parinfo(order)

int             order;  Placement in list of parameters to
                        type-check
```

Description:

_parinfo() is used to test the type of a passed parameter. _parinfo(0) returns the number of parameters passed and _parinfo(n) returns the type of parameter where n is the position in the parameter list. The value returned is one of the following:

```
/* _parinfo types from extend.h*/
#define     UNDEF       0
#define     CHARACTER   1
#define     NUMERIC     2
#define     LOGICAL     4
#define     DATE        8
#define     MPTR        32      /* or'ed with type when
                                   passed by reference */
#define     MEMO        65
#define     ARRAY       512
```

To check that a parameter has been passed by reference, add its type value to MPTR(32) and test for that sum. If a character string is passed by reference, _parinfo(n) should equate to 33.

Example: The following example illustrates how to check for reference-passing.

In Clipper:

```
CLEAR
X = "STRING"
UDF(@X)
UDF(X)
```

In C:

```
#include "extend.h"

CLIPPER udf()
{

   printf("%s %s PASSED BY REFERENCE\r\n", _parc(1),
   ISBYREF(1) ? "IS" : "IS NOT");

   \*

   The code above is the same as:

   printf("%s %s PASSED BY REFERENCE\r\n", _parc(1),
   _parinfo(1) == CHARACTER + MPTR ? "IS" : "IS NOT");
   *\

   printf("_parinfo(1) = %d\r\n\r\n", _parinfo(1));
}
```

Output:

STRING IS PASSED BY REFERENCE
_parinfo(1) = 33

STRING IS NOT PASSED BY REFERENCE
_parinfo(1) = 1

Example:

The following user-defined C function, cfunc(), receives four parameters from Clipper: a character type, a numeric, a logical, and a date; defines the C variables to receive the values, then validates the parameters.

```
CLIPPER cfunc()          /* no formal parameters */
{
char          *parm1;
double        parm2;
int           parm3;     /* logical is declared
                            as int */
char          *parm4;    /* date declared as
```

```
                                    char (YYYYMMDD) */

if (PCOUNT == 4  && ISCHAR(1) && ISNUM(2)
                 && ISLOG(3) && ISDATE(4))

    {
        <code to execute parameters valid>
    }
    else
    {
        <code to execute parameters invalid >
    }

}
```

_parl()

Passes Clipper logical as int

Syntax Usage:

```
#include        <extend.h>

int             _parl(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parl() gets a logical parameter from Clipper and converts it to int where (1 = .T. and 0 = .F.).

_parnd()

Passes a Clipper numeric as double

Syntax Usage:

```
#include        <extend.h>

double          _parnd(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parnd() gets a numeric parameter from Clipper converting it to double.

_parni()

Passes Clipper numeric as int

Syntax Usage:

```
#include        <extend.h>

int             _parni(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parni() gets a numeric parameter from Clipper converting it to int.

_parnl()

Passes Clipper numeric as long

Syntax Usage:

```
#include        <extend.h>

long            _parnl(order [, index])

int             order;      Placement in actual parameter list
int             index;      Array element index
```

Description:

_parnl() gets a numeric parameter from Clipper converting it to long.

_ret()

Return to Clipper

Syntax Usage:

```
#include        <extend.h>

void            _ret(void)
```

Description:

_ret() has no Clipper return value. This is useful so you can execute a C function using the DO command as if it were a Clipper procedure.

_retc()

Pass a character string to Clipper

Syntax Usage:

```
#include        <extend.h>

void            _retc(string)

char            *string;  Pointer to return string
```

Description:

_retc() passes to Clipper a character pointer for the character string you want to pass back to your application program.

_retclen()

Pass a character string and its specified length to Clipper

Syntax Usage:

```
#include        <extend.h>

void            _retclen(string, len)

char            *string;  Character string to count
int             len;      Length of string, including embedded
                          CHR(0)
```

Description:

_retclen() returns a string with a specified length. This function differs from _retc() in that it accounts for embedded CHR(0)s within the string.

_retds()

Pass a date string to a Clipper date

Syntax Usage:

```
#include        <extend.h>

void            _retds(string)

char            *string; Date string in the form (YYYYMMDD)
```

Description:

_retds() passes to Clipper a character pointer to a string in the form YYYYMMDD as date type .

_retl()

Pass an int to a Clipper logical

Syntax Usage:

```
#include        <extend.h>

void            _retl(flag)

int             flag;    Boolean value
```

Description:

_retl() passes an int to Clipper as a logical value, where 1 is true (.T.) and 0 is false (.F.).

_retnd()

Pass a double to a Clipper numeric

Syntax Usage:

```
#include        <extend.h>
void            _retnd(n)
double          n;   Numeric expression
```

Description:

_retnd() passes a double to Clipper as numeric type.

_retni()

Pass an int to a Clipper numeric

Syntax Usage:

```
#include        <extend.h>
void            _retni(n)
int             n;   Numeric expression
```

Description:

_retni() passes an int to Clipper as a numeric integer.

_retnl()

Pass a long to a Clipper numeric

Syntax Usage:

```
#include        <extend.h>
void            _retnl(n)
long            n;   Long numeric expression
```

Description:

_retnl() passes a long integer to Clipper as numeric type.

INTERFACING WITH ASSEMBLY LANGUAGE

In addition to supporting a mixed language interface to C and other high-level langauges, Clipper also supports user-defined functions written in assembler through the Extend System. Like the interface to C, the Extend System provides access to the parameter passing and returning functions. These are the same functions available from C differing only by the method of access. Here you must manipulate the stack in order that the proper values are known when Clipper is called. To make this process easier and more rational, two sets of macros are included (EXTENDA.MAC and EXTENDA.INC).

EXTENDA.MAC is similar to earlier versions of Clipper and is provided in Summer '87 as a means of retaining compatibility with assmbler code written for Autumn '86. EXTENDA.INC is a new set of macros developed for use with Summer '87 and Microsoft MASM version 5.0.

Passing Parameters

These are functions used for obtaining the parameters being passed from your Clipper code. When you use the __PAR functions, follow the pseudo sequence below:

- Move the parameter number to be obtained into a register
- Push the register
- Call the appropriate __PAR function for the data type of the parameter
- The parameter is received in registers, either by value or by pointer
- Restore the stack

In addition, it is important to remember that Clipper is written in C and all parameters are passed as C data types.

SAMPLE ASSEMBLY ROUTINES

To interface an assembly language routine to Clipper using the Extend System requires that you follow some specific rules.

- Declare your routine PUBLIC.

- Declare the Extend functions to be used as EXTRN and FAR or INCLUDE EXTENDA.INC which makes these declarations for you.

- If you define your own data segment, group it together with Clipper's DGROUP. Otherwise, DS must point to DGROUP before you call any Extend function.

- Class your data segment as DATA for Autumn '86 and Summer '87.

- Class your code segment PROG for Autumn '86 or CODE for Summer '87.

- If you have not passed a value back to Clipper, call __RET just before your assembler routine ends.

The following is a shell of a typical Clipper user-defined function written in assembly language that demonstrates these rules:

```
PUBLIC          <func_name>

EXTRN           <exten_func>:FAR

DGROUP    GROUP <data_seg>

<data_seg>      SEGMENT PUBLIC  'DATA'
;
;   <your data declarations >
;
<data_seg>      ENDS

<code_seg>      SEGMENT 'CODE'
                ASSUME cs:<code_seg>,  ds:DGROUP

<func_name>  PROC   FAR
                push   bp      ; Save registers
                mov    bp,sp
                push   ds
                push   es
                push   si
                push   di

                <your code goes here>
```

```
                    pop     di      ; Restore registers
                    pop     si
                    pop     es
                    pop     ds
                    pop     bp
<func_name>         ENDP            ; End of routine

<code_seg>          ENDS            ; End of code segment
                    END
```

Example:

The following is an operational assembly language routine that clears a region of the screen.

In Clipper:

```
Clearit(10, 10, 20, 60)
```

In assembly language:

```
PUBLIC Clearit                      ; Declare as public.

EXTRN       _ _PARNI:FAR            ; Declare functions
EXTRN       _ _RET:FAR              ; as external.

DGROUP GROUP      DATASG            ; Combine your data
                                    ; segment with Clipper's.

DATASG      SEGMENT    'DATA'       ; Start of data segment.
top         DB      0
left        DB      0
bottom      DB      0
right       DB      0

DATASG      ENDS                    ; End of data segment.

_PROG       segment        'CODE' ; Start of code segment.
               ASSUME     cs:_PROG,ds:DGROUP

CLEARIT        PROC    FAR          ; Start of process.

               push    bp           ; Preserve return address.
               mov     bp,sp

               push    ds           ; Save registers.
               push    es
```

```
push    si
push    di

mov     ax,1        ; Point to parameter.
push    ax          ; Place on stack.
call    _ _PARNI    ; Call Clipper.
add     sp,2        ; Restore stack.
mov     top, al     ; Assign parameter
                    ; to top.

mov     ax,2        ; Repeat process for next
                    ; parameter.
push    ax
call    _ _PARNI
add     sp,2
mov     left, al    ; Assign to left.

mov     ax,3        ; Repeat process for next
                    ; parameter.
push    ax
call    _ _PARNI
add     sp,2
mov     BOTTOM,al   ; Assign to bottom.

mov     ax,4        ; Repeat process for next
                    ; parameter.
push    ax
call    _ _PARNI
add     sp,2
mov     right, al   ; Assign to right.

mov     ch, top     ; Place coordinates in
                    ; CX:DX.
mov     cl, left
mov     dh, bottom
mov     dl, right

mov     ax, 0600h   ; Request roll up service.
mov     bh, 07      ; Normal attribute.
int     10h         ; Issue video interrupt.

pop     di
pop     si
pop     es          ; Restore registers.
pop     ds
pop     bp

call    _ _RET      ; Clipper return (actual
                    ; cleaning).
```

```
                ret                 ; Actual physical return.
CLEARIT         ENDP                ; End of process.

_prog           ENDS
                END
```

Since assembly language gives you access to the lowest level of software and hardware interaction, you must exercise appropriate caution. All the functions above require that you save the registers before you can use them to call the Extend functions. Be certain that after calling any Extend function you restore the stack by incrementing the stack pointer (SP). Failure to restore the original environment will usually cause the system to crash.

ASSEMBLY LANGUAGE EXTEND MACROS

A macro in assembler serves a similar function to macro substitution in Clipper. Essentially you substitute a symbolic name for the text of command statements. In Clipper, you can substitute macros for expressions and literal constants. In assembler, the implementation is much broader. You can substitute macros for entire blocks of code and, additionally, pass parameters to the macro code. When MASM encounters the macro reference at assembly time, it substitutes the associated code for the macro name and the text of the parameter name with the actual parameter information.

Macros can be defined within the source file in which they are used or within INCLUDE files. INCLUDE files are the most practical place to store macros since this allows you to isolate code that is volatile or environment specific.

EXTENDA.MAC

EXTENDA.MAC is provided in Clipper Summer '87 to give compatibility with assembler routines that were developed with Autumn '86.

> **Warning:** EXTENDA.MAC macros do not save registers used automatically. You must, therefore, save registers of importance to you before calling any of these macros.

EXTENDA.MAC contains the following macros:

Table 11-4 EXTENDA.MAC Macros

Macro	Function
GET_PCOUNT	Number of parameters passed in AX
GET_PTYPE	Type of parameter passed in AX
GET_CHAR	Address of a string in AX:BX
GET_INT	Integer in AX
GET_LONG	Long integer in AX:BX
GET_DBL	Double in AX:BX:CX:DX
GET_DATESTR	Address of date string in AX:BX
GET_LOGICAL	Logical in AX
RET_CHAR	Returns string pointed to
RET_INT	Returns integer
RET_LONG	Returns long integer
RET_DBL	Returns double
RET_DATESTR	Returns date string
RET_LOGICAL	Returns logical value

EXTENDA.INC

A new system of assembly language macros is included to facilitate the use of assembly language within Clipper applications. These macros require an assembler compatible with Microsoft MASM 5.0. To use them, place the following directive at the beginning of the Assembler source file:

INCLUDE EXTENDA.INC

Using this macro system, a user-defined function written in assembly language has the following general format:

```
    CLppublic      list_of_UDFs

;*******
;
    CLfunc function_type function_name  [parameter_list]
    CLcode
    .
    . body of function
    .
    CLret      return_value
```

More specifically, and to illustrate actual syntax:

```
   CLpublic <CRYPT, FUNC1, FUNC2, FUNC3, FUNC4>

;*******
;
;   str = CRYPT(str, len)
;   encrypt/decrypt a character string
;   CLfunc char   CRYPT  <char str, int len>

   CLcode
      PUSH      ES

   ; test for valid parameter
      CMP       PCOUNT, 2
      JB        CRYPT_RET
      TESTNUL   str
      JZ        CRYPT_RET

   ; parameters acceptable
      LES       SI,    str
      MOV       BX, 0

CRYPT_LOOP:
      CMP       BX, len
      JE        CRYPT_RET
      NOT       BYTE   PTRES:[BX + SI]
      INC       BX
      JMP       CRYPT_LOOP

CRYPT_RET:
      POP       ES
   CLret        str
```

The angle brackets, <>, are required by MASM. The use of mixed upper and lower case is necessary only if the file is to be assembled with case sensitivity enabled. We will assume that case sensitivity IS enabled for the remainder of this discussion.

The Four Basic Macros

There are many other macros in the package, but the four described below represent a minimum for getting started, so take a close look at them and the concept in general.

```
CLpublic <CRYPT, FUNC1, FUNC2, FUNC3, FUNC4>
```

Every function to be called from within Clipper must be declared public in this way. Note that function names are upper case and separated by commas.

```
CLfunc char      CRYPT  <char str, int len>
```

This is the function declaration. It tells the macro system several things about the function including the return type, function name, and parameter list. Note that each parameter in the list is declared to be of a particular type. This list must be omitted completely if there are no parameters.

```
CLcode
```

This is implemented as a separate step because some of the optional macros are allowed, or required, to appear between the function declaration and the start of the function. The actual code for setting up a user-defined function is placed here. Everything necessary for the Clipper interface (including the fetching of parameters and run-time type checking) is generated by this macro. When this code is executed at runtime, it stores PCOUNT (the Clipper parameter COUNT) so that it may be accessed at any point in the function. Additionally, parameters that are not supplied and parameters of the wrong data type are set to zero (NUL). The macro TESTNUL can be used to test for any null parameter, and should always be used to test for a null pointer.

```
CLret     str
```

This macro generates the cleanup code for a proper return to Clipper. It performs an assembly-time type check of the return value based on the function type declared in CLfunc. If a function is declared as type "int," the return value can be any 16-bit register (i.e., "CLret CX"), or any variable of type "int." Similarly, a function declared as type "char" will return a pointer to a string. In this case, the return value can be any two 16-bit registers (i.e., "CLret BX DX"), or any variable of type "char." Although functions declared "void" have no return value, the CLret macro must still be called but without the return value.

Macros designed to appear in assembler source code are listed below with syntax and some detail. See EXAMPLEA.ASM for more examples. Also, many of these macros call each other within the include file itself.

EXTENDA.INC

```
CLpublic <FUNC1, FUNC2, ...>
```

Declares one or more functions public. All functions to be called from within a Clipper application must be declared public in this way. Function names must be upper case.

```
CLfunc ftype NAME <ptype p1, ptype p2, ...>
```

Declare a function. NAME must be upper case. The function type determines how the specified value will be returned to Clipper in the subsequent call to CLret. Allowable function types (ftype) are char, int, long, double, log, date, and void. The void type is for functions that return no value at all. Allowable parameter types (ptype) are char, int, long, double, log, and date.

```
CLcode
```

This macro must be called before the body of the function. At runtime, PCOUNT is set to the number of incoming parameters. Missing parameters and parameters of the wrong data type are set to zero.

```
CLret r
```

Type checking is enforced according to the return type specified in the function declaration (CLfunc). Allowable return values (r) include any variable of the correct type, one or two 16-bit registers depending on the size of the declared function type, or nothing if the function has been declared void.

```
PCOUNT
```

Set by CLcode to the number of incoming parameters. The value stored in PCOUNT can be accessed at any point in the function.

```
TESTNUL variable
```

Sets the zero flag if variable is zero (NUL). This macro can be used to test any size variable, but must be called to test pointers.

```
CODESEG segname
```

The default code segment is filename_TEXT where filename represents the base name of the source file. If this macro is used to override the default, the code segment will be declared as segname_TEXT instead. This macro must appear before any code-generating macros that would cause the default code segment to be declared (i.e., CLfunc or WORKFUNCS).

```
DATASEG segname
```

This macro works much the same as CODESEG except that the DATA segment is being declared. The default data segment is data. Unlike CODESEG, however, DATASEG is not called automatically. It is not necessary to call this macro unless there is static data to be declared, in which case DATASEG must be called before CLstatic. The segname parameter is optional.

```
CLstatic <stype v1 i1, stype v2 i2, ...>
```

Declares static data where v is a variable and i is an initial value to be placed in v. Allowable static types (stype) are byte, len, int, log, long, double, cptr, and string. Several of these types will be better understood with further explanation.

Len is used to determine the size of the data block between the current address and i, where i is a previously defined label. For example:

```
CLstatic <byte msg1 "This is a test...">
CLstatic <len msglen1 msg1>
CLstatic <byte msg2 "this is ONLY a test.">
CLstatic <len msglen2 msg2, len msglen3 msg1>
```

This defines the following values:

```
msglen1 = size of msg1
msglen2 = size of msg2
msglen3 = msglen1 + msglen2.
```

The cptr type causes the additional symbols v_OFF and v_SEG to be defined. These symbols facilitate access to just the segment or just the offset portion of a pointer. Static pointers can be initialized (at assembly time) to point to static data.

The string type is much the same as byte except that a null byte is added at the end of the string.

```
CLlocal <type l1, ltype l2, ...>
```

Declares auto (stack based) data. This macro can only appear between CLfunc and CLcode. Allowable local types (ltype) are char, int, long, double, log, and date.

```
CLextern <xtype x1, xtype x2, ...>
```

Declares external code and data. Allowable external types (xtype) are byte, int, log, long, double, cptr, and far. Far is used to declare external functions. Cptr is used to access external pointers in the form (segment:offset).

```
CLlabel <ltype l1, ltype l2, ...>
```

Label a memory address using the assembler LABEL pseudo op. Allowable label types (ltype) are byte, int, log, long, double, cptr, and string. The cptr type causes the additional symbols l_OFF and l_SEG to be defined. These symbols facilitate access to just the segment or just the offset portion of a pointer.

```
Cglobal <v1, v2, ...>
```

Use this macro to make static data available to other modules. Data declared global cannot be accessed directly by a Clipper application, but other modules written in C or assembly language can share data this way.

```
Ccall cfunc <param1, param2, ...>
```

Call a C function. This macro will handle all parameters and stack manipulation to interface with the C language. A parameter can be any 16-bit register, a defined variable, or a constant (immediate) value. For example, if cfunc has been declared external with CLextern:

```
Ccall cfunc <str, AX, i, 5>
```

```
WORKFUNCS
```

Begin local assembler routines. This macro handles the segment declarations only. Within the WORKFUNCS block, standard assembly language practice should be observed. Procedures must be declared explicitly and ended with the ENDP directive.

ENDWORK

End local assembler routines. This macro must be called to terminate the WORKFUNCS block. If WORKFUNCS is followed by CLfunc, ENDWORK will be called automatically. However, if WORKFUNCS is at the end of the file, ENDWORK must be called explicitly.

SES reg16

The compliment of LES. Stores ES and a named register to a DWORD memory operand. For example:

```
SES    DI, pointer_variable
```

SDS reg16

The compliment of LDS. Stores DS and a named register to a DWORD memory operand. For example:

```
SDS    SI, pointer_variable
```

DOSREQ number

Calls DOS interrupt 21H with the request specified by number. All necessary setup (other than loading the AH register) must be performed before issuing this call.

$DEFINE symbol value

Makes assembler equate look like a C #define and may be used instead of the assembler statement "symbol EQU value".

OFFPART p

The first word of a DWORD memory operand (segment:offset)

SEGPART p

The second word of a DWORD memory operand (segment:offset)

LSW n

The first word of a DWORD memory operand (32-bit number)

MSW n

The second word of a DWORD memory operand (32-bit number)

> Note: The END directive that tells MASM when to stop is not generated by any of the macros and must be entered explicitly.

EXTEND SYSTEM ASSEMBLY LANGUAGE FUNCTIONS

__PARC

Syntax Usage:

1. To obtain a string parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARC
add     sp,2
```

2. To obtain a string element from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARC
add     sp,4
```

Arguments:

<ORDER> is the placement order in the parameter list.
<INDEX>is the index of the array element to be accessed.

Description:

This routine places the address of a Clipper character string in DX:AX where:

DX = segment
AX = offset

__PARCLEN

Syntax Usage:

1. To obtain length of a string parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARCLEN
add     sp,2
```

2. To obtain length of a string element from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARCLEN
add     sp,4
```

Arguments:

<ORDER> is the placement order in the parameter list.
<INDEX> is the index of the array element to be accessed.

Description:

__PARCLEN returns the length of a given string in AX. It is used primarily with respect to CHR(0). This has two uses:

- Check for the length of a string with embedded CHR(0)s in it
- Get the length of a string without counting the null terminator

__PARCSIZ

Syntax Usage:

1. To obtain size of memory allocated for a string parameter:

```
mov     ax,<ORDER>
push    ax
call    __PARCSIZ
add     sp,2
```

2. To obtain size of memory allocated for a string element from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARCSIZ
add     sp,4
```

Arguments:

<ORDER> is the placement order in the parameter list.
<INDEX> is the index of the array element to be accessed.

Description:

__PARCSIZ returns the number of bytes in memory allocated for the specified string, including the null terminator, in AX. Note that __PARCSIZ returns zero for constants, e.g., DO <process> WITH "ABC". The parameter must be passed by reference with the "@" symbol.

__PARDS

Syntax Usage:

1. To obtain a date parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARDS
add     sp,2
```

2. To obtain a date element from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARDS
add     sp,4
```

Arguments:

<ORDER> is the placement order in the parameter list.
<INDEX> is the index of the array element to be accessed.

Description:

This routine places the address of a Clipper date stored as a string in the form YYYYMMDD in DX:AX where:

DX = segment
AX = offset

__PARINFA

Syntax Usage:

1. To obtain number of elements in array parameter (length of array):

```
mov     ax, <ORDER>
mov     bx, 0
push    bx
push    ax
call    __PARINFA
add     sp,2
```

2. To obtain type of element of the array parameter:

```
mov     ax, <ORDER>
mov     bx,  <INDEX>
push    bx
push    ax
call    __PARINFA
add     sp,4
```

Argument:

<ORDER> is the placement order in the parameter list.
<INDEX> is the index of array element to be type-checked.

Description:

If the <INDEX> argument is zero, __PARINFA places the length of the array passed in AX. If the <INDEX> argument is greater than zero, __PARINFA places the type of the specified parameter in AX using the following values:

Table 11-5 AX Values (__PARINFA)

Type	Value placed in AX	
undefined	0	
character	1	
numeric	2	
logical	4	
date	8	
by reference	32	; OR with type
memo	65	
array	512	

__PARINFO

Syntax Usage:

1. To obtain number of parameters passed:

```
mov     ax, 0
push    ax
call    __PARINFO
add     sp,2
```

2. To obtain type of parameter passed:

```
mov     ax, <ORDER>
push    ax
call    __PARINFO
add     sp,2
```

Argument:

<ORDER> is the placement order in the parameter list.

Description:

If the <ORDER> argument is zero, __PARINFO places the number of parameters passed in AX. If the <ORDER> argument is greater than zero, __PARINFO places the type of the specified parameter in AX using the following values:

Table 11-6 AX Values (__PARINFO)

Type	Value placed in AX	
undefined	0	
character	1	
numeric	2	
logical	4	
date	8	
by reference	32	; OR with type
memo	65	
array	512	

To check that a parameter has been passed by reference, add its type value to MPTR(32) and test for that sum. If a character string was passed by reference, __PARINFO(N) should equate to 33.

__PARL

Syntax Usage:

1. To obtain a logical parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARL
add     sp,2
```

2. To obtain a logical element from an array parameter:

```
mov     ax, <ORDER>
mov     bx,<INDEX>
push    bx
push    ax
call    __PARL
add     sp,4
```

Arguments:

<ORDER> is the placement order in the parameter list.
<INDEX> is the index of the array element to be accessed.

Description:

This routine places the word of a Clipper logical value in AX where 1 is true (.T.) and 0 is false (.F.).

__PARND

Syntax Usage:

1. To obtain a numeric double parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARND
add     sp,2
```

2. To obtain a numeric double element from an array parameter:

```
mov     ax,<ORDER>
mov     bx, <INDEX>
push    ax
push    ax
call    __PARND
add     sp,4
```

Arguments:

<ORDER> is the placement order in a parameter list.
<INDEX> is the index of array element to be accessed.

Description:

This routine places the address of double passed from a Clipper numeric in DX:AX where:

DX = segment
AX = offset

To get the value in AX:BX:CX:DX, you would:

```
mov     ax, <ORDER>
push    ax
call    __PARND
add     sp,2
mov     es,dx
mov     si,ax
mov     ax,es:[si]
mov     bx,es:[si + 2]
mov     cx,es:[si + 4]
mov     dx,es:[si + 6]
```

__PARNI

Syntax Usage:

1. To obtain a numeric integer parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARNI
add     sp,2
```

2. To obtain a numeric integer from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARNI
add     sp,4
```

Arguments:

<ORDER> is the placement order in a parameter list.
<INDEX> is the index of array element to be accessed.

Description:

This routine receives a Clipper integer number and places the value in AX.

__PARNL

Syntax Usage:

1. To obtain a numeric long parameter:

```
mov     ax, <ORDER>
push    ax
call    __PARNL
add     sp,2
```

2. To obtain a numeric long from an array parameter:

```
mov     ax, <ORDER>
mov     bx, <INDEX>
push    bx
push    ax
call    __PARNL
add     sp,4
```

Argument:

<ORDER> is the placement order in the parameter list.
<INDEX> is the array element to access.

Description:

This routine receives a Clipper number as a long and places the value in DX:AX.

__RET

Syntax Usage:

call __RET

Description:

For "DO" routines that do not return values.

__RETC

Syntax Usage:

```
mov     <register1>,    <SEGMENT>
mov     <register2>,    <OFFSET>
push    <register1>
push    <register2>
call    __RETC
add     sp,4
```

Arguments:

 is the segment address of string.

<OFFSET> is the offset address of string.

Description:

__RETC passes back to Clipper a character string pointed to by <register1> and <register2>.

__RETCLEN

Syntax Usage:

```
mov     <register1>, <SEGMENT>
mov     <register2>,  <OFFSET>
mov     <register3>, <LENGTH>
push    <register3>
push    <register1>
push    <register2>
call    __RETCLEN
add     sp,6
```

Arguments:

  is the segment address of string.

<OFFSET> is the offset address of string.

<LENGTH> is the length of the string to return to Clipper.

Description:

__RETCLEN passes back to Clipper a character string pointed to by <register1> and <register2>. It passes it with the specified length. This string can have embedded CHR(0)s.

__RETDS

Syntax Usage:

```
mov     <register1>, <SEGMENT>
mov     <register2>, <OFFSET>
```

```
push    <register1>
push    <register2>
call    __RETDS
add     sp,4
```

Argument:

 is the segment address of date string.

<OFFSET> is the offset address of date string.

Description:

__RETDS passes back to a Clipper date a string in the form (YYYYMMDD).

__RETL

Syntax Usage:

```
mov     <register1>, <expN>
push    <register1>
call    __RETNL
add     sp,2
```

Argument:

<expN> is the value to return to a Clipper logical value where 0 is false (.F.) and 1 is true (.T.).

Description:

Passes back to Clipper a word value as a logical value.

__RETND

Syntax Usage:

```
mov     <register1>, <expN1>
mov     <register2>, <  ?    >
mov     <register3>, <  ?    >
mov     <register4>, <  ?    >
push    <register1>
push    <register2>
```

```
push    <register3>
push    <register4>
call    __RETND
add     sp,8
```

Arguments:

<expN1> is the value to return to Clipper.

Description:

__RETND passes back to Clipper a double precision number to numeric.

__RETNI

Syntax Usage:

```
mov     <register1>, <expN1>
push    <register1>
call    __RETNI
add     sp,2
```

Argument:

<expN1> is the value to be returned as an int.

Description:

__RETNI passes back to Clipper a number stored in <register1> as a numeric.

__RETNL

Syntax Usage:

```
mov     <register1>,<expN1>
mov     <register2>, <  ?     >
push    <register1>
push    <register2>
call    __RETNL
add     sp,4
```

Argument:

<expN1> is the long value to return to Clipper.

Description:

__RETNL passes back to Clipper a long integer stored in the register pair <register1>: <register2> to a numeric.

SUMMARY OF CHANGES FROM AUTUMN '86

This section briefly summarizes the differences between the Autumn '86 version of Clipper and the Summer '87 version for both the C and Assembler Extend System interfaces.

The C Interface

1. C Extend functions must now be defined using the CLIPPER macro or declaring the function "void pascal." For example:

```
Autumn '86:   c_routine()
              {
              ...
              }

Summer '87:   CLIPPER c_routine()
              {
              ...
              }
```

2. Extend functions must all now be preceded with an underscore (_) character. For example:

```
Autumn '86:   parni(1)
              parinfa(2, 1)
              ret()

Summer '87:   _parni(1)
              _parinfa(2, 1)
              _ret()
```

3. The _parc() function points to the original string.

4. There are three new functions:

_parclen()
_retclen()
_parcsiz()

Note: Clipper strings are still null terminated.

The Assembler Interface

1. References to Extend functions must now be preceded with an extra underscore (_) character.

```
Autumn '86:   call _PARN
              call _PARC
              call _RETNI
              call _RET
Summer '87:   call _ _PARNI
              call _ _PARC
              call _ _RETNI
              call _ _RET
```

2. For some of the _ _PAR functions, the receiving registers and how they are received, has changed.

_ _PARC

Autumn '86: Received parameter in AX:BX (segment:offset) by value.

Summer '87: Receives the passed parameter in DX:AX (segment:offset) by reference. This function points to the original string.

_ _PARNL

Autumn '86: Received parameter in AX:BX passed by value.

Summer '87: Receives parameter in DX:AX passed by value.

_ _PARND

Autumn '86: Received parameter in AX:BX:CX:DX passed by value.

Summer '87: Receives parameter in DX:AX (segment:offset) passed by reference.

__PARDS

Autumn '86: Received parameter in AX:BX (segment:offset) passed by reference.

Summer '87: Receives parameter in DX:AX (segment:offset) passed by reference.

3. There are three new functions:

   __PARCLEN()
   __PARCSIZ()
   __RETCLEN()

Note: Clipper strings are still null terminated.

4. Data segments should be classed "DATA". Code segments should be classed "CODE".

Note: Some of the macros in EXTENDA.MAC have also changed. If you Include this file, please examine the changes first.

12 Clipper Utility Programs

Chapter 12 contains instructions for the Clipper utility programs. These programs have been included so you can create or modify the files and to assist you during the development of your programs. Each utility is described below:

- The DBU.EXE program allows you to create, browse, view, and index database files without writing separate programs or using an interpreter.
- The RL.EXE program is used to create or modify a report form file or to create or modify a mailing label form file.
- The LINE program is used to display or print your programs with line numbers.
- The MAKE.EXE program (Based on the Unix Make utility) is used to describe source and object file dependencies to the Clipper compiler and to streamline the compile and link cycle.

THE DBU.EXE PROGRAM

DBU is a multi-purpose utility for data and file manipulation. It is a tool for accomplishing the "side work" that is a necessary part of application development. It is distributed as source code which must be compiled and linked with Clipper.lib and Extend.lib to create DBU.EXE. MAKEDBU.BAT is provided to create DBU.EXE.

The syntax for creating DBU.EXE is:

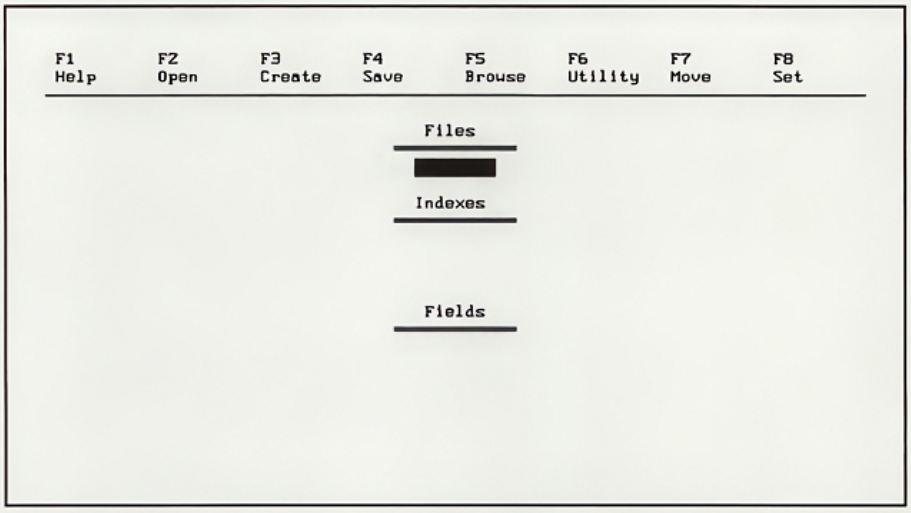
C>MAKEDBU

In DBU there are two innovative concepts that work cooperatively: a menuless program structure and a global data View. Together, these two concepts form a coherent and useful package.

After a data View is set, most user activity takes place in the Browse, Create, and Index modes, always under control of the "menuless program" umbrella. Keystroke usage is logically consistent across these activities, and context sensitive help is always available.

The Menuless Program

The user interface is simple and intuitive and requires very little typing. The top two lines of the screen always contain a list of active function keys like this:



Pressing the appropriate function key will activate pull-down menus. You can only highlight and select options available in the current environment. For example, the Index option of the Create mode will only be available if a database file has been opened. Available options are displayed in high intensity.

It is no longer necessary to return to a main menu to make a new selection. In other words, the user can go directly to any area of the program at any time with just one keystroke.

Keystrokes

The user interface is simple and consistent. Function keys pull down a list of options for each mode. Cursor keys navigate between modes and choices in lists. The Insert, Delete, and Enter keys are used to add, delete, or replace items on the screen. When adding or replacing, a list of items is presented from which a selection can be made. When a list is not practical, an entry field is placed on the screen. In either case, pressing the Enter key will confirm the selection or entry; pressing the Escape key will cancel it. If there is no selection or entry pending, pressing the Escape key will exit the program. The Delete key is used to remove items from lists and close files.

Mode Options

F1 Help

Help - Context sensitive help

F2 Open

Database - Select a .dbf file
Index - Select an .ntx file
View - Select a .vew file

F3 Create

Database - Create or modify a .dbf file
Index - Create a new .ntx file

F4 Save

View - Save the current environment in a .vew file
Structure - Save the newly created or modified .dbf file

F5 Browse

Database - Browse the currently selected .dbf file
View - Browse the currently selected set of .dbf files

F6 Utility

Copy - Copy the current .dbf file to a new file
Append - Append records into the currently selected file from another file.
Pack - Pack the current file
Zap - Remove all records from the current file
Run - Execute another application from within DBU. (Run "COMMAND" to invoke DOS without terminating the current process. "EXIT" will return you to DBU from DOS.)

F7 Move

Seek - Move to record with matching index key expression
Goto - Move to record pointer
Locate - Move to record that matches expression
Skip - Move to next record

F8 Set

Relation - Set relation between two open database files in the current view
Filter - Set filter to records that meet the specified condition
Fields - Add fields to current view

The Global Data View

The global data View is a particular configuration of data files, index files, field lists, relations, and filters. The program normally starts in View mode. Here, data and index files may be opened or closed, field lists may be altered, and relations and filters may be set. Once established, the View remains in effect until it is deliberately altered by the user. The general idea is that the various parts of the program will use the current view to direct their activities.

The View can be quite complex. As many as six work areas can be active at the same time and multi-child relations are fully supported. With many open files it is often desirable to suppress several fields from each file. Setting up a complex View can require substantial time and effort. In order to simplify this

process, the current View can be saved in a .vew file for fast restoration. Once restored, small modifications are quick and simple. You may pass the name of a .vew file that you saved during a previous session on the command line to DBU.

Browse

Browse allows alteration of the data in the current View. All fields listed in the current View are displayed on a single row, one row per record. Pressing the Enter key will select a field for editing.

Press the insert key to append a new record.

Cursor movement in Browse mode is very flexible. Paging, panning, beginning and end of file, extreme left and right pan are all available. Additionally, the "Move" key allows access to the search functions SEEK, GOTO, SKIP and LOCATE, enabling the rapid location of a particular record in a large data file.

Create

Create allows you to create or modify database file sructures, or create index files. In either case, the new structure may be saved under any name. If a file with the same name already exists, MODIFY STRUCTURE is assumed and the program will attempt to preserve any data in the existing file.

Creating or modifying index files is very easy with DBU. The key expression can be read from any index file and then altered and/or used to create or alter another or the same index file. When the names of the index files and the key expression are all correct, a single keystroke triggers the generation of the index file.

Help

The help key is always active. Context sensitive help is available from all parts and all levels of the program.

THE RL.EXE PROGRAM

The RL utility allows you to create or modify existing report form (.frm) and label form (.lbl) files. Report and label files modified using this utility program are used by the REPORT FORM and LABEL FORM commands to specify screen and printer output. RL is distributed as source code which must be compiled and linked with Clipper.lib and Extend.lib to create RL.EXE. MAKERL.BAT is provided to create RL.EXE.

The syntax for creating RL.EXE is:

C>MAKERL

Starting RL.EXE

Begin using RL to create or modify report and label files by entering the following command at the DOS prompt:

C>RL

Creating Or Modifying A REPORT FORM File

To create a new REPORT FORM file or to modify an existing file, press the Enter key with "REPORT" highlighted. A file selection box will appear. Select a file by highlighting and pressing the Enter key or enter the name of the new file you wish to create. Highlight "Ok" and press any key to begin designing the report. Highlight "Cancel" and press any key to return to the RL start-up screen. The field definitions screen will be displayed.

```
F1      F2      F3      F4      F5      F6      F7      F10
Help    Layout  Groups  Fields  Delete  Insert  Go To   Exit

                                                File SMPL_RPT.FRM
                                                        Field 1
                    — Field Definitions —               Total 1

Contents

Heading
     1
     2
     3
     4

Formatting
Width      10
Decimals    0
Totals      N
```

Across the top of the screen is a list of the activities you may choose while designing a REPORT FORM. Press the appropriate function key to select one of the options.

Report Options

F1 Help

Context sensitive help

F2 Layout

By selecting layout, you may specify the page header, page width, margins, page length, line spacing, and printer directives.

F3 Group

Group specifications allow you to decide which fields will cause a break in the report when a new value is encountered. You may specify the group expression, a heading to display when a new value is found, whether the report will display summary information only, and whether to eject after each group. You may also define sub-groups and sub-group headings.

F4 Field Definitions

This is the default screen when creating reports. The field definition is the expression you wish to display in the column for each record in the database file. You can also specify a column heading, the width of the column including decimals, and whether or not to total the column. Use the PgUp and PgDn keys to move from column to column.

F5 Delete

Pressing function key 5 allows you to delete the current column.

F6 Insert

Pressing function key 6 allows you to insert a new column at the current position.

F7 Go To

Pressing function key 7 allows you to select a column number to view and edit.

F10 Exit

Press function key 10 to exit the report option of RL. Highlight "Ok" to save and exit, "No" to exit without saving your changes, or "Cancel" to resume defining your REPORT FORM. Press any key to select your highlighted choice.

Creating Or Modifying A LABEL FORM File

To create a new LABEL FORM file or to modify an existing file, press the Enter key with "LABEL" highlighted. A file selection box will appear. Select a file by highlighting and pressing the Enter key or enter the name of the new file you wish to create. Highlight "Ok" and press any key to begin designing the label. Highlight "Cancel" and press any key to return to the RL start-up screen. The label definition screen will be displayed. Across the top of the screen is a list of the activities you may choose while designing a LABEL FORM.

```
F1        F2        F3                                        F10
Help      Toggle    Formats                                   Exit
                                                   File SMPL_LBL.LBL
 Dimensions                    Formatting
      Width      35                  Margin      0
      Height      5                  Lines       1
      Across      1                  Spaces      0

      Remarks

            CONTENTS
Line 1
```

Label Options

F1 Help

Context sensitive help

F2 Toggle

Pressing function key 2 allows you to switch between label dimensions and label contents.

F3 Formats

Pressing function key 3 allows you to select from a table of common label formats.

NOTE: Everything following a comma on each line of the label will be ignored, however, any valid expression is allowed.

THE INDEX PROGRAM

Included with Clipper is an indexing program that has been written to assist in creating indexes for your database files.

The index program included with Clipper (Index.prg) is supplied as a program file that you must compile and link before use.

The program file is provided, rather than an executable load module, so that you can see some of the techniques used in Clipper programs.

The INDEX utility program provides you with the ability to select a particular database and create a Clipper index file. When you have compiled and linked the Index.prg program, enter the following command:

C>INDEX <d:database filename>

Where: "d:" is the disk drive identification letter.

"Filename" is the name of the database file that you wish to create an index file for.

You do not need to include the filename extension (.dbf).

The INDEX program will display the different fields contained in the database and ask you to enter the index filename and the field(s) that you wish to index on.

THE LINE PROGRAM

When a program is compiled, errors are displayed on the screen or, if you have specified, captured in a file. Each error message shows the line number of the corresponding statement in your program.

To assist in finding the erroneous statements in the program, Clipper includes a Utility program which will produce line numbers for the statements in your program (LINE.EXE).

To display the line numbers in your program enter the following command and press Enter.

C>LINE <programname.prg> [n]

Where: "Programname" is the name of your program file (you MUST enter the .prg filename extension)

"n" is an optional numeric parameter entered if you wish to display a particular line from one of your source code files.

If you enter a number greater than 5, this option displays the specified line along with the previous 5 lines and 10 lines following. Entering 1 for this option will pause the display every twenty two lines. The following example will display lines 95 through 110 of Dbu.prg:

C>LINE Dbu.prg 100

You may use DOS redirection to print the file or to write the output to another file.

C>LINE Dbu.prg > PRN

THE MAKE PROGRAM

Using Make with Clipper

When developing a large Clipper application, it is customary to divide the program up into a number of separate .prg files which can be compiled separately and combined by the linker. This way, when changes are made, only the files changed need to be re-compiled. Keeping track of what needs to be re-compiled can become tedious. How can you be sure that the .OBJ file matches the .prg?

The Clipper Make utility allows you to define a number of source code files and how they are to be compiled and linked. Make then builds the system for you, only re-compiling and linking when necessary.

How Make Works

To use Make, you first create a "description file" that tells Make which files comprise the system, and how they are to be compiled and linked. For each .OBJ file, it is necessary to specify the source file it depends upon, and how the source file is to be translated to create that target .OBJ file. For each executable (.EXE) file it is necessary to specify what .OBJ files it depends upon, and how the .OBJ files are to be linked to create this target executable file. A number of rules are specified, which define what dependent files a target is contingent upon, and how the target is built from its dependents.

When Make is invoked, it reads the file and uses the rules to determine what needs to be done to build the system. It does this by comparing the dates and times of the target files against the dates and times of the dependent files. If any of the dependent files have a more recent time and date stamp than the object file, they are re-compiled according to the instructions for this rule. If none of the dependents of a particular target have a more recent time stamp than that target, the commands for that rule are skipped over.

Make places the commands that need to be performed in a batch file which it invokes when it is finished.

Make Description File

The description file is a regular text file consisting of one or more rules. The rules have the following format :

```
<targetfile : <dependentfiles >>
    <command1>
    <command2 >
    <more commands>
```

Only one target file can be specified per rule, but multiple dependent files and commands may be specified. Each command must be on a separate line. The "/" character can be used to continue a line.

For example, assume TEST1.OBJ is dependent upon Test1.prg, Test2.prg and Test3.prg. To create this file, you run Clipper on test1, using Clipper's ability to compile referenced programs to automatically compile test2 and test3. The rule for this is:

```
Test1.obj : Test1.prg Test2.prg Test3.prg
    Clipper Test1
```

This tells Make that TEST1.OBJ is the target, and that it is dependent upon Test1.prg, Test2.prg and Test3.prg. To make TEST1.OBJ, it is necessary to compile Test1 with the command "Clipper Test1".

To create the .EXE file, Test1 is linked with a module called Windows, as well as the Clipper and Extend libraries. The rule for this is:

```
Test1.exe : Test1.obj Windows.obj
    LINK Test1 Windows,,,\Clipper\Clipper  \Clipper\Extend
```

Order of Rules

The order in which these rules appear in the description file is very important. Make processes the rules one at a time in the order they are encountered, and if one rule causes a file to be updated which is a dependent of a target in an earlier rule, this earlier rule will not be re-processed. For example, assume the two sample rules just given appear in the description file in the following order:

```
Test1.exe : Test1.obj Windows.obj
    LINK Test1 Windows,,,\Clipper\Clipper \Clipper\Extend

Test1.obj : Test1.prg Test2.prg Test3.prg
    Clipper Test1
```

If Test2.prg has just been updated and needs to be re-compiled, Make first examines the rule with TEST1.EXE as the target. It finds that TEST1.EXE is up to date with regard to TEST1.OBJ and WINDOWS.OBJ, and that no action needs to be taken. It then processes the second rule and finds that Test2.prg has a more recent time and date stamp than TEST1.OBJ, and it therefore needs to perform the commands associated with this rule, i.e. Clipper Test1. Make finds no more rules to process and ends. To bring the .EXE file up to date, it is necessary to perform the link.

The statements should appear like this:

```
Test1.obj: Test1.prg Test2.prg Test3.prg
    Clipper Test1

Test1.exe: Test1.obj Windows.obj
    LINK Test1 Windows,,,\Clipper\Clipper   \Clipper\Extend
```

Starting Make

To invoke Make, simply type "Make" followed by the name of the description file. If the above description file is called TEST1.MAK, enter:

```
C>Make test1.mak
```

Make only accepts one option, /N. This directs Make to display the commands that would be invoked, but does not actually execute them. The following command runs Make in this manner.

```
C>Make /N test1.mak
```

Comments and Macros

The pound sign (#) character is used to define a comment. It can be used as the first character on a line, in which case the entire line is a comment, or it can be used following a command.

```
Test1.exe : Test1.prg Test2.prg Test3.prg
# This is a comment
# so is this line
```

Make allows you to define macros to be used within the description file. A macro allows you to associate a string of characters with a given name. When it subsequently encounters the name, it is replaced with the character string. A macro definition command takes the following form :

name = value

To subsequently use name as a macro, it must be preceded by a dollar sign ($) and enclosed in parentheses. For example:

```
# define files macro
files = Test1.obj Test2.obj Test3.obj

Test1.exe: $(files)

# that line was identical to
# test1.exe: test1.obj test2.obj test3.obj
```

A SAMPLE SYSTEM

The sample system consists of the following source files:

Driver.prg	Co_query.prg
Screen.prg	Co_kill.prg
Utils.prg	Co_select.prg
View.prg	Co_dbfs.prg
Command.prg	Windows.asm
Co_edit.prg	

To build this system, the .prg files are compiled with Clipper, and the .ASM file assembled with MASM. This produces one .OBJ file for each source file. Link these .OBJ files with the Clipper and Extend libraries to create an executable file called QBE.EXE.

Here is the complete description file for this sample system.

```
# macros
obj1 = Driver.obj View.obj Command.obj Screen.obj Utils.obj Co_edit.obJ
obj2 = Co_query.obj Co_kill.obj Co_select.obj Co_dbfs.obj Windows.obj

#PRGs
Driver.obj: Driver.prg
      Clipper Driver -m

Screen.obj: Screen.prg
      Clipper Screen -m

Utils.obj:  Utils.prg
      Clipper Utils -m

View.obj :  View.prg
      Clipper View -m

Command.obj:Command.prg
      Clipper Command -m

Co_edit.obj: Co_edit.prg
      Clipper Co_edit -m

Co_query.obj: Co_query.prg
      Clipper Co_query -m

Co_kill.obj: Co_kill.prg
      Clipper Co_kill -m

Co_select.obj: Co_select.prg
      Clipper Co_select -m

Co_dbfs.obj: Co_dbfs.prg
      Clipper Co_dbfs -m

# ASM
Windows.obj: Windows.asm
      MASM Windows;

# EXE
Qbe.exe: $(obj1) $(obj2)
      LINK @Qbe.lnk,Qbe,,\Clipper\Clipper   \Clipper\Extend/SE:256
```

Note the definition and use of the two macros, obj1 and obj2.

Inference Rules

It is tedious to have to write the same Clipper command on every rule with .prg files as the dependents. Usually when converting a particular source language file to an object file, the same command is used. It is rare, for example, to use different compiler flags on different files. In the sample system, the -m flag is used on every compile. To overcome this inconvenience, Make provides what are known as "Inference rules". These rules direct Make to produce a set of files with one extension from a set of files with another. For example, we can tell it how to convert .prg files to .OBJ files, or .ASM files to .OBJ files. Specify a generic rule that is applied every time it encounters a rule with the specified target and dependent extensions.

In the sample system, to define an inference rule that creates .OBJ files from .prg files, it is necessary to invoke the command :

Clipper <filename> -m

You do not need to repeat this command under each .OBJ - .prg dependency.

An inference rule takes the following form:

```
.<dependentextension>.<targetextension>:
    <command1>
    <command2>
    .
    .
```

Since these rules are generic (they apply to any files with the specified extension), they cannot refer to filenames directly. You need to use a predefined special macro ($*), which means the base name portion (the filename without the extension) of the target of the particular rule being applied. The inference rule to create .OBJ files from .prg files is:

```
.prg.obj:
    Clipper $* -m
```

The description file for the sample system can now be written as :

```
# macros
obj1 = Driver.obj View.obj Command.obj Screen.obj Utils.obj Co_edit.obj
obj2 = Co_query.obj Co_kill.obj Co_select.obj Co_dbfs.obj Windows.obj

# PRG files
.prg.obj:
	CLIPPER $* -m

# ASM files
.asm.obj:
	MASM $*;

#PRGs
Driver.obj	: Driver.prg
Screen.obj	: Screen.prg
Utils.obj	: Utils.prg
View.obj	: View.prg
Command.obj	: Command.prg
Co_edit.obj	: Co_edit.prg
Co_query.obj	: Co_query.prg
Co_kill.obj	: Co_kill.prg
Co_select.obj	: Co_select.prg
Co_dbfs.obj	: Co_dbfs.prg

# ASM
Windows.obj :Windows.asm

# EXE
Qbe.exe		: $(obj1) $(obj2)
	LINK @Qbe.lnk,Qbe,,\Clipper\Clipper  \Clipper\Extend/se:256
```

There are two other predefined special macros that Make recognizes. These are:

$@ : The name of the target file including the extension.

$** : The complete list of dependencies for this target.

The $@ macro is useful for displaying the name of the file being processed. For example, you can write the inference rule for .prg files as:

```
.prg.obj:
    REM     *** $@   Clipper $* -m
```

When processed , the $@ is expanded to be the complete filename of the particular .OBJ file being produced.

The $** macro is useful on the link line. For example :

```
Qbe.exe: $(obj1) $(obj2)
    Link $**,$*,,\Clipper\Clipper  \Clipper\Extend/se:256
```

The $** is expanded to the complete list of dependencies, as defined by the two macros, obj1 and obj2.

THE SWITCH PROGRAM

SWITCH.EXE is a utility used to chain from one executable file to another. The executable files may be Clipper-compiled applications or other programs such as word processors or spreadsheets.

SWITCH is particularly useful in those cases where memory is insufficient to use the RUN command from a Clipper program, or when the other memory requirements of a Clipper program are too large for a particular system. In the latter case, the program can be broken into several independent executable modules.

The Path of Execution

To execute SWITCH, you specify SWITCH and a list of executable programs separated by spaces.

For example:

C>SWITCH SHELL AR AP GL UTIL

During start-up, SWITCH assigns a sequential number to each parameter. The first parameter is assigned a position number of 0, the second parameter is assigned a position number of 1, and so forth to the end of the parameter list.

Each Clipper program in the parameter list can direct execution to another program through a DOS exit code set by ERRORLEVEL(). If an exit code is not set, DOS sets the exit code to zero and execution returns to the first program in the parameter list.

At start-up, SWITCH executes the first program specified in the parameter list. Thereafter, SWITCH executes the program whose sequential position in the parameters list equals the indicated exit code.

Parameters should be separated by spaces only. Any other delimiter aborts execution entirely. The SWITCH utility itself and batch files may not be used as one of the parameters of SWITCH.

Exiting SWITCH

You must use an exit code of zero from program zero (typically a menu) to terminate SWITCH.

Example

The following example demonstrates the use of SWITCH:

```
* Menu.prg
INPUT "Press 0 to exit, 1 to execute" +;
   "Myprog1, 2 to execute Myprog2" to Choice
ERRORLEVEL(choice)
RETURN

* Myprog1.prg        RETURNS to Menu
? "My program 1."
WAIT
ERRORLEVEL(0)
RETURN

* Myprog2.prg        RETURNS to Myprog1
? "My program 2."
WAIT
ERRORLEVEL(1)
RETURN
```

The following command:

C> SWITCH MENU MYPROG1 MYPROG2

sequentially assigns the numbers 0, 1 and 2 to Menu, Myprog1 and Myprog2 respectively and then executes the programs in the following order when the user enters "2" in response to the first prompt, and "0" in response to the second prompt:

```
SWITCH
Menu       && User enters 2.
Myprog2
Myprog1
Menu       && User enters 0.
```

Entering zero at program zero returns you to DOS.

Passing Parameters

To pass information between Clipper-compiled applications, use memory files.

APPENDICES

The Appendix contains information that is necessary for a complete understanding of Clipper. Topics covered include:

- Appendix A - Explains how to obtain technical support and assistance in using Clipper.
- Appendix B - Contains the Nantucket Software License Agreement.
- Appendix C - Contains a list of dBASE III PLUS commands not supported by Clipper.
- Appendix D - Contains error messages displayed by Clipper during the compilation process.
- Appendix E - Contains error messages displayed by the PLINK86-Plus Linker.
- Appendix F - Contains error messages displayed during the execution of the program.
- Appendix G - Contains an ASCII chart and a list of INKEY() values.
- Appendix H - Contains the Reserved Word List.
- Appendix I - Lists the PLINK86-Plus commands.
- Appendix J - Contains programming examples.

APPENDIX A

Nantucket Support

Technical Support And Assistance

Nantucket Corporation provides technical support to registered users according to the following plans. Support includes responses to technical inquiries submitted by telephone, letter, telex, or through the TechMail Forum in the Nantucket Private Network on The Source.

30 DAYS FREE SUPPORT: Support is free to registered users for 30 days after Nantucket receives your Licensee Registration Card.

PER-INQUIRY FEE: You may charge inquiries to your MasterCard, VISA or American Express card at $25 (U.S. currency) for each inquiry answered. Be sure to include your name as it appears on your card, the card type, account number, and expiration date. All non-phone inquiries should include your System Disk 1 serial number. Direct them to the attention of Nantucket Support.

YEARLY SUPPORT SUBSCRIPTION: For a yearly fee of $99 (U.S. currency), Nantucket offers Clipper users an annual subscription that covers the software developer's complete needs. The Support subscription includes:

Technical Support
Unlimited inquiries may be submitted by telephone, letter, telex, or through the TechMail Forum in the Nantucket Private Network on The Source. A technician will answer your questions regarding Clipper and will help with your programming needs. Phone calls may be placed to our Support group Monday through Friday from 8:00 a.m. to 4:30 p.m. Pacific Time. (Please note, however, that support closes at 4:00 p.m. on Tuesdays.) All non-phone inquiries should include your System Disk 1 serial number. Direct them to the attention of Nantucket Support.

Nantucket News
Support subscribers also receive an annual subscription to Nantucket News, Nantucket's quarterly journal. Please see below for more information.

On-line Information

Nantucket provides on-line technical information through The Source Information Network.

The Nantucket Special Interest Group (NANSIG) is available to all members of The Source. NANSIG embodies an extensive program and file library, a user bulletin board and helpful programming tips.

Registered Clipper users have access to the Nantucket Private Network (NPN), the private section of NANSIG. Here you will find technical support, complete anomaly reports, reference notes and special forums. You may also participate in Nantucket-sponsored conferences.

A modem, communications software and a membership to The Source allow immediate access to NANSIG. To gain NPN access privileges, send your name, address, registered Clipper serial number, and The Source ID number with your request for access to the NPN SysOp at SourceMail NAN002.

Nantucket News

Nantucket News is the quarterly technical journal published by Nantucket Support. It contains the latest information about Nantucket products including programming techniques, reported anomalies with work-arounds, application notes and news from NANSIG and NPN on The Source.

The newsletter is sent without additional charge to all Nantucket Support subscribers. You may also purchase an annual subscription, without Support, for $25 in the U.S.; elsewhere for $35 (U.S. currency).

To subscribe to Nantucket Support or Nantucket News, fill out and mail the enclosed card. Nantucket Support can be reached by calling the dedicated Support line at (213)827-8664.

APPENDIX B

Nantucket Software License Agreement

Software License

This software product is licensed to you for your personal use. You may use it on any single machine. HOWEVER, IT MAY NOT BE USED BY MORE THAN ONE PERSON NOR ON MORE THAN ONE MACHINE SIMULTANEOUSLY. YOU MAY NOT NETWORK THE PRODUCT.

Your are permitted to make up to four copies solely for archival or backup purposes.

You may transfer the program on a permanent basis and license it to another party if the third party agrees to accept the terms and conditions of this agreement. If you transfer the program, you must at the same time either transfer all copies of the program regardless of form to the same party or destroy any copies not transferred.

You may utilize the program to prepare derivative works of other programs which you own or control with no further license from or payment to Nantucket, provided Nantucket's program cannot be separated out, in whole or in part, from any such derivative works.

Term

This license shall be effective from the date you open this package and remain in effect for a period of twenty-five (25) years unless terminated sooner as provided in this agreement. You may terminate this license at any time by destroying all copies of the program (in any form). Your license terminates automatically if you fail to comply with any terms or conditions of this agreement.

Limited Warranty

Nantucket warrants that the program will conform, as to all substantial operational features, to Nantucket's current published specifications, documentation and authorized advertising; that, the user documentation accompanying the program contains the necessary information to utilize the program; and that, the media on which the program is furnished shall be free from defects in materials and workmanship for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your sales receipt. EXCEPT FOR THE LIMITED WARRANTY SET FORTH

ABOVE, THE PROGRAM IS PROVIDED "AS IS". NANTUCKET MAKES NO OTHER WARRANTY, EXPRESS OR IMPLIED, WITH RESPECT TO THE PROGRAM AND SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. NANTUCKET DOES NOT WARRANT THAT THE PROGRAM WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE PROGRAM WILL BE UNINTERRUPTED OR ERROR FREE. YOU ARE SOLELY RESPONSIBLE FOR THE SELECTION OF THE PROGRAM TO ACHIEVE YOUR INTENDED RESULTS AND FOR THE RESULTS ACTUALLY OBTAINED.

Some states do not allow the exclusion or limitation of implied warranties, so the above exclusions and limitations may not apply to you. This warranty gives you specific rights and you may also have other rights which vary from state to state.

Limitation of Remedies and Liability

In the event the program fails to meet Nantucket's "limited warranty", Nantucket's entire liability and your exclusive remedies shall be:

A. The replacement of any program not meeting Nantucket's "limited warranty" which is returned to Nantucket or any authorized dealer with a copy of your sales receipt.

B. If Nantucket cannot or will not replace the program, you may terminate this Agreement by returning the program and all copies to Nantucket and your money will be refunded.

Under no circumstances, and notwithstanding any failure of the essential purpose of any limited remedy provided for herein, shall Nantucket be liable to you for any damages, claims or losses whatsoever, including but not limited to any claims for lost profits, lost savings or other special, incidential or consequential damages arising out of the use or inability to use the program regardless of the circumstances.

EXCEPT AS EXPRESSLY PROVIDED FOR IN THIS LICENSE, YOU MAY NOT USE, COPY, MODIFY, REPRODUCE, TRANSFER OR DISTRIBUTE THE PROGRAM IN WHOLE OR IN PART, WITHOUT FIRST OBTAINING NANTUCKET'S WRITTEN PERMISSION.

This agreement will be governed by the laws of the State of California.

NANTUCKET CORPORATION

APPENDIX C

dBASE III PLUS Commands and Functions not Supported by Clipper

Some dBASE III PLUS commands and functions are not supported by Clipper. Clipper does not support any of the commands that are used primarily in the interactive or "dot prompt" mode. In the interactive mode, you may instantly query the various databases without writing a program, however, Clipper has been designed to compile and execute programs significantly faster than can be accomplished in the interactive mode.

The dBASE III PLUS commands and functions that are not supported by Clipper are listed in the table below.

Table C-1 dBASE III PLUS Commands and Functions not Supported by Clipper

APPEND	IMPORT FROM
ASSIST	INSERT
BROWSE	LIST FILES
CHANGE	LIST HISTORY
CLEAR FIELDS	LIST MEMORY
CREATE LABEL	LIST STATUS
CREATE REPORT	LIST STRUCTURE
CREATE QUERY	LOAD
CREATE SCREEN	LOGOUT
CREATE VIEW	MESSAGE
DISPLAY FILES	MODIFY COMMAND
DISPLAY MEMORY	MODIFY LABEL
DISPLAY STATUS	MODIFY QUERY
DISPLAY STRUCTURE	MODIFY REPORT
DISPLAY USERS	MODIFY SCREEN
EDIT	MODIFY STRUCTURE
ERROR()	MODIFY VIEW
EXPORT TO	ON ERROR/ESCAPE/KEY
HELP	RESUME

RETRY	SET HELP
RETURN TO MASTER	SET HISTORY
SET	SET MEMO WIDTH
SET CARRY	SET MENUS
SET CATALOG	SET SAFETY
SET COLOR ON/OFF	SET STATUS
SET DEBUG	SET STEP
SET DO HISTORY	SET TALK
SET ECHO	SET TITLE
SET ENCRYPTION	SET TYPEAHEAD
SET FIELDS	SET VIEW
SET HEADING	

Many of the operations of dBASE commands not directly supported by Clipper are handled through various Clipper Utility programs. See Chapter 12 for more information on these equivalent operations.

APPENDIX D

Clipper Compiler Error Messages

As Clipper compiles your programs, any errors found are displayed on your screen (or in a file if you have redirected the compiler output).

The error display identifies the program, the program line number, and the syntax of the line containing the error.

The following messages are displayed when the compiler detects improper usage of a command.

ACCEPT/INPUT error
APPEND error
ASSIGNMENT error
AVERAGE error
AVERAGE number of expressions and variables
CALL error
CLEAR error
CLOSE error
COPY error
COUNT error
CREATE error
DELETE error
DO error
FILTER error
FOR error
LABEL error
LIST error
LOCATE error
PARAMETER error
RECALL error
RELATION error
RELEASE error
REPLACE error

RESTORE error
SAVE error
SET error
SET not recognized
SET switch error
SUM error
USE error
WAIT error

The following messages relate to specific errors or situations.

Case w/o do case	Preceding DO CASE not found.
else w/o if	Preceding IF <condition> not found.
endcase w/o do case	Preceding DO CASE not found.
enddo w/o while	Preceding DO WHILE not found.
endif w/o if	Preceding IF <condition> not found.
exit w/o do while	Preceding DO WHILE <condition> not found.
Fatal at <n> - invalid procedure mode	Two procedure files contain the same procedure or have procedures with the same name. Note: This error message is displayed on your screen only; it is not redirected to a file that you have created to list error messages.
illegal device	SET DEVICE TO <screen/print> - illegal option.
illegal lvalue	The left side of an assignment statement is not a legal name. You cannot STORE to an expression (as in store 12 to 100 + 1). You cannot assign a value to a function (as in INKEY () = 12).

loop w/o do while	LOOP must be within a DO WHILE...ENDDO structure.
missing 2nd quote	Terminating string delimiter not found.
next w/o for	NEXT must have corresponding FOR.
otherwise w/o do case	OTHERWISE must be within a DO CASE...ENDCASE structure.
rest of line ignored	Syntax of the remainder of the line not supported - ignored by compiler.
symbol redefinition error	All files and procedures must have unique names. This message is usually displayed when a procedure file and a procedure name within it have been given the same name.
too many constants	The compiler has encountered too many constants while parsing the source code. Compile the program in sections and combine with the linker.
too many symbols	The compiler has encountered too many symbols while parsing the source code. Compile the program in sections and combine with the linker.
unbalanced DO CASE	Matching ENDCASE not found. Can be caused by any unbalanced conditional statement.
unbalanced DO WHILE	Matching ENDDO not found. Can be caused by any unbalanced conditional statement.
unbalanced FOR/NEXT	NEXT for previous FOR not found. Can be caused by any unbalanced conditional statement.

unbalanced IF/ELSE	ENDIF for previous IF not found. Can be caused by any unbalanced conditional statement.
verb not recognized	Command not supported by Clipper. Check command spelling and syntax.

APPENDIX E

PLINK86-Plus Link Editor Error And Warning Messages

During the linking process, errors or potential runtime problems are discovered by the PLINK86-Plus Linker. When an error or potential runtime problem is detected, an error or warning message is displayed on the screen. In addition, error or warning message code numbers are displayed. Following is a list of the codes and descriptions for error and warning messages.

If you experience one of the following warning/error messages and are unable to resolve the problem, DO NOT contact Phoenix Technologies, Ltd.

WARNING MESSAGES

Occasionally PLINK86-Plus detects a situation that may cause a problem during the execution of the application load module.

Warning Code	Description
1	An attempt was made to use a 16 bit address to reference a location that is more than 64k bytes away, or to use an 8 bit address to reference a location more that 256 bytes away. Another possibility is that the address being referenced is lower in memory than the segment register that is assumed. For example, a near call to a memory address lower than the current CS, or a far call that is using the wrong segment address. The offset actually used will be "wrapped around" to a number within range, so the program probably will not address the object correctly.
2	Under DOS, the desired stack location of a program is kept in the .EXE file header. This is used to set the SS and SP registers when the program is executed. PLINK86-Plus looks for a stack segment marked as such

by the compiler or assembler being used. If found, its eventual address is placed into the header. If not found, zeroes are used and the program probably will not function correctly unless it sets the SS and SP registers itself. Even then, the program could crash if an interrupt occurs before it sets up a valid stack.

3 A GROUP is a collection of segments that must reside within a 64K memory space. This allows 16 bit addresses to be used to access objects within the group. Part of the group cannot be accessed if the group is too large and you must, therefore, reduce the size of the group. Sometimes this warning may appear even when you are positive that the segments making up the group total less than 64K in size. Look in the memory map to see if the segments have been separated by intervening segments not in the group.

This situation often occurs when attempting to mix assembly language modules with those from a high level language. The class and segment names you use in the assembly language should match those used in the high level language. If they do not match, your assembler segments will probably be assigned memory locations at the end of the section and may be too far away from the other members of the group.

Hint: Overlays may be used to decrease the memory requirements of a code group.

4 The module named was used in the MODULE command, but PLINK86-Plus did not find a module of that name in the input files.

5 The 8086 processor uses a dual addressing scheme. The offset portion of a long address is relative to the physical 64K segment selected by the paragraph portion.

Therefore, although the offset can be determined at linkage edit time, the paragraph address is an absolute address in the physical memory and must be adjusted according to where the operating system loads the program in memory. DOS does this as the program is loaded for execution. PLINK86-Plus outputs a long address relative to the start of the program and continues linking.

6 This warning should not occur using DOS (only occurs using CP/M-86).

7 The named module contains a record type unfamiliar to PLINK86-Plus. The whole record will be skipped. These messages are inhibited after a few have been printed. This usually occurs when you have included the .prg filename extension when identifying your program name to a DOS batch file (CL.BAT).

8 Each record in an object file contains a check field at the end for validation purposes. This message indicates that the checksum was bad, but linking continues. These messages are inhibited after a few have been printed. If the object file was patched onto the disk before the link, check the checksum. Some library managers also seem to be lax about making sure the checksums are correct. If the file is really smashed, a fatal error will probably occur soon after this message.

9 Each record in an object file is preceded by a word giving the record size (it may be revealed by the DUMP utility). This error means that PLINK86-Plus reached the end of the record and found that the number of bytes processed is different from the specified size. The object file is probably smashed but PLINK86-Plus will attempt to continue processing it.

10 A 16 bit address is being used while the target object is more than 64K bytes away from the frame when referenced from the named module (or an 8 bit address is used with a distance greater than 256 bytes). The target, therefore, cannot be accessed as desired. The address actually used will be "wrapped around" to fit into the required offset size. If there is a group larger than 64K, it will have to be made smaller for this access to be made correctly.

11 There may be only one definition for each global (PUBLIC) symbol in the program being linked. Another definition was found for the named symbol. PLINK86-Plus ignores the duplicate definition, retains the first one, and continues linking.

12 The named public segment was assigned to more than one group. One module placed the segment in one group, and later another module specified a different group for the same segment. References to the segment may be calculated incorrectly.

13 The named segment was first defined as a PUBLIC segment, and then later redefined as a common block (or visa versa). Make sure all definitions of the segment are changed to the same type. This problem usually occurs when linking assembly language modules.

14 A duplicate stack segment was defined within the named module. PLINK86-Plus uses the last stack definition made to specify the stack in the .EXE header. If this stack segment is empty or too small, the program will probably crash soon after execution. Since the previous stack segment is ignored, there is no point in having more than one stack segment. Make sure that all stack segment definitions use the same name and class name. They will then be combined

into one segment having as its size the sum of the sizes of the original segments.

19 Not enough memory for file path. PLINK86-Plus will prompt for a path name of a file it cannot locate. If there is not enough memory to save this string, warning 19 will occur and pass 2 will prompt for the path again.

20 Bad allocation size in section #_. If the calculated section size is not the same as the actual section size, warning 20 occurs.

21 Bad fixup count in section #_. If the pass 1 fixup count is not the same as the pass 2 fixup count, warning 21 occurs.

ERROR MESSAGES

Command Syntax Errors

These errors are caused by mistakes made in the input information given to PLINK86-Plus. The program line number causing the problem will be displayed with a couple of question marks inserted after the point where the error was detected. These should aid in locating the problem. Occasionally PLINK86-Plus may not detect the error until more text is processed. If the question marks appear at the beginning of a line, check the end of the previous line for errors.

Error Code	Description
1	"@" files are nested too deeply. Only three levels of "@" files may be active at any given time. Check also for a loop in the "@" file references.
2	Disk error encountered while reading "@" file. Try rebuilding the file.
3	PLINK86-Plus cannot find the file named after an "@".
5	The item given for input at this time is too large. The maximum size allowed is 64 characters.
6	Invalid digit in number. Valid digits depend on the radix being used (the default is hexadecimal for addresses, decimal for all else).
10	Invalid filename. The input stream should contain a valid filename for the DOS operating system.
11	Expecting a statement. A keyword that begins a statement should be present here.
14	Expecting an identifier. A section, module, segment, or symbol name must be entered at this point.

15	Expecting an equal (=) sign.
16	Expecting a value. A 16-bit quantity must appear at this point.
17	No files were given to link. You must use the "FILE" statement and specify at least one input file.
18	Expecting a right parenthesis sign at the end of the CLASS statement. The list of segment names must be enclosed in parentheses.

Work File Errors

When PLINK86-Plus runs out of memory it opens a work file on disk named PLINK86.WRK to hold the description of the program. The following error codes indicate a problem with the processing of the work file.

Error Code	Description
30	The work file cannot be created. There is probably no space in the disk directory.
31	An I/O error occurred while writing the work file.
32	An I/O error occurred while reading the work file.
33	An I/O error occurred while repositioning (doing an lseek on) the work file.
34	There are too many objects (symbols, segments, groups, etc.) defined in this program (approximately 35,000 may be created). The program is too large for PLINK to handle.

Input Object File Errors

The following errors have to do with the object files that are given to PLINK86-Plus to link. Usually these errors occur when a file has been somehow corrupted. Try recompiling to get a new copy of the object file. If it is a library supplied by the compiler manufacturer, try to get a fresh copy.

Error Code	Description
41	Premature end of object file. The end of the indicated file was reached unexpectedly. Possibly the file was truncated by copying it with a program that assumes that a CNTL -Z (1AH) is the end of the file. This usually occurs when you have included the .prg filename extension when identifying your program name to a DOS batch file (CL.BAT).
42	Fatal read error in object input file.
43	PLINK86-Plus could not locate the named object file. When an object file cannot be located, PLINK86-Plus will normally ask the operator for a filename prefix such as a drive or path name (see FILE statement). If the operator elects to terminate the linkage edit, this error will be displayed. It will also be displayed immediately if the BATCH command was used; the operator is not prompted in this case.

Output File Errors

The following errors are caused by a problem in creating the output code file or memory map file (when written to disk). Often these errors are caused by a full disk or disk directory, a disk that is write-protected, or some kind of hardware problem with the disk.

Error Code	Description
45	Cannot create the output disk file. The disk directory may be full or the disk write-protected.
46	Invalid output file type. If given, the file extension must be .EXE or .CMD.
47	Fatal disk write error in output file. The disk may be full or write-protected, or a hardware error has occurred.
48	Fatal disk read error in output file. An irrecoverable hardware error has probably occurred.
49	Cannot close output file. The disk may be write-protected or a hardware error has occurred.
50	Cannot create the memory map disk file. The disk directory may be full or the disk write-protected.

Miscellaneous Errors

Error Code	Description
51	Undefined symbols exist. You may have misspelled a procedure or a function. The listed symbols were in one or more modules, but were never defined. They will have to be defined as internals of some module.
54	There is not enough memory on your machine to run PLINK86-Plus. PLINK86-Plus requires at least a 256K system to run.
57	There is a problem with the OVERLAY.LIB file. Make sure you are using the most recent version. If you created your own OVERLAY.LIB, be sure it is properly formatted.
58	The stack segment is too large (greater than 64K bytes). Remember that the stack segments defined in each module are concatenated together as are public segments.

Intel Format Object File Errors

These errors are caused by a problem with the format of the input object files. These should be in the format defined by Intel Corporation compiler output.

It is possible that the relocatable object file has, in some way, become corrupted. Try recompiling the program.

Error Code	Description
61	An LTL segment appeared in an Intel module. PLINK does not accept these as input.
62	A REGINT record specified a register to be initialized in a way unsupported by the .EXE file format. Only CS:IP and SS:IP (the program starting address and stack pointer) may be pre-initialized with a REGINT record.
63	A LIDATA sub-record has a repeat count of zero.
64	An Intel format object library file was used that has an improperly formed library index. It may help to rebuild the library using PLINK86-Plus.
65	An absolute starting address was specified in the given module. PLINK86-Plus requires that the starting address be given relative to some segment.
66	This module uses a group element type not handled by PLINK86-Plus. Currently only segments may be included in groups (group component description code = FFH).
70	An invalid location was specified for a fixup. The LOC field is greater than 4 for a segment relative fixup.
71	An invalid location was specified for a fixup. The LOC field is greater than 1 for a self-relative fixup.

73	A frame specification type unsupported by PLINK86-Plus (5 or 7) was used.

Program Structure Errors

These errors involve a problem with the overlay structure that was specified. Correct the problem and re-link.

Error Code	Description
80	Overlays are nested too deeply. There are too many levels in the overlay structure.
81	There are too many ENDAREA statements - more than the number of BEGINAREA statements that have not been closed out yet.
82	The BEGINAREA and ENDAREA statements are unbalanced. There should be the same number of each.
83	The program being linked will not fit into a 1 megabyte address space which is the maximum supported by the Intel 8086 microprocessor family. You will have to use overlays to reduce the memory requirements of the program.

PLINK86-Plus Diagnostic Errors (Error Codes 200 thru 226)

These errors indicate a problem with PLINK86-Plus itself. You have NOT caused the problem, and it is unlikely that you can fix the problem. First, try running PLINK86-Plus again; if that does not work, recopy the PLINK86-Plus program from your Clipper diskette and run it again. If the error persists, record the error code and contact Nantucket (DO NOT contact Phoenix Technologies, Ltd.).

APPENDIX F

Application Runtime Error Messages

APPENDIX F

The Runtime Error System

In Clipper Summer '87, there are two categories of runtime errors: recoverable and non-recoverable. Recoverable errors are divided into classes with each class having an associated error recovery function. The classes are as follows:

- Database error
- Expression error
- Miscellaneous error
- Print error
- Open error
- Undefined error

OVERVIEW OF ERROR FUNCTIONS

The default low-level error handling functions supplied in CLIPPER.LIB are written in Clipper. The source code for these functions is provided in the file Errorsys.prg. If you wish to supply more extensive runtime error handling you may modify or rewrite the functions in Errorsys.prg. The new functions can be compiled with Clipper and linked within your application as replacements for the default functions in CLIPPER.LIB.

Also included are debugging versions of the low-level error function in Alterror.prg. Each of the functions in this file provide facilities that are useful during the debugging phase of a program.

To substitute new low-level error functions for those found in CLIPPER.LIB, you can link a new ERRORSYS.OBJ, ALTERROR.OBJ, or a new error function by itself. The only proviso is that you link your main program first before any of the object files.

Automatic Parameters

All of the low-level error handling functions share a common parameter structure. When one of the functions is called, it is as if the following code has been executed:

```
xxx_error(name, line, info [, model [, _1 [, _2;
    [, _3 [, _4 [, _5]]]]]])
```

All of the functions are called with the "name," "line," and "info" parameters. The other parameters may or may not be passed to a particular error function.

Name Parameter

The "name" parameter contains a character type value. The value is equal to the name of the procedure or function which was executing when the error occurred.

Line Parameter

The "line" parameter contains a numeric value. The value is equal to the source code line number of the statement which caused the error. Note that this line number is relative to the beginning of the source file, not the start of the function or procedure. If the source file was compiled with the -l option (line numbers disabled), this parameter will contain zero.

Info Parameter

The "info" parameter contains a character type value. The value consists of information about the error. The information consists of a descriptive phrase (e.g., "Type error") optionally followed by information describing the situation of the error (e.g., ("In macro")). The additional information, if supplied, is surrounded by parentheses.

Model Parameter

The "model" parameter, if supplied, contains a character string. The value consists of a fragment of Clipper source code illustrating the operation which caused the error. Note that the fragment is not identical to the original source code which caused the error. The fragment can be used with the operand parameters (discussed below) to reconstruct the operation which caused the error.

Operand Parameter

The "_1," "_2," etc. parameters, if supplied, may contain values of any type, depending on the circumstances of the error. These values are equal to the values which were supplied to the Clipper operation which failed.

Note that, although they are unusual, the names "_1," "_2," etc. are valid Clipper identifier names. These names were chosen to avoid conflicts with identifiers in your application.

Recovery Strategy

Using the automatic parameters passed, you can construct a strategy to recover from these errors. In addition to the automatic parameters there are several support functions you can use. They are as follows:

PROCNAME()

You can use this function to identify, display, or pass as an argument to another user-defined function, the name of the error function called.

DOSERROR()

DOSERROR() returns the current error number. You will find this function useful for determining the cause of a RUN error. The DOS error will tell you whether the problem lies with the location of COMMAND.COM or with there not being enough memory to RUN the specified program.

ALTD()

This function invokes the Clipper Debugger. There are two modes appropriate for use within an error function. The first, and most commonly used in the default error functions, is ALTD(2) which invokes the Debugger displaying the Variable:View Privates screen. The second mode, and most useful for interactive debugging, is ALTD() with no argument. Invoking the Debugger in this mode displays the last screen accessed. Ideally, you will have set the screen you want to display next and then resume execution of your program using the Alt-G speed key to Control:Go.

BEGIN SEQUENCE...BREAK...END

This command is a control structure you can use to pass control from an error function to a local error handler. The basic idea is to block the commands that are most likely to generate a runtime error (device and printer errors) with the BEGIN SEQUENCE...END control structure statements. Then place the local error handler statements immediately following the

END statement. In the appropriate error function, place a BREAK where you want control to branch to the END statement in the procedure that initiated the error.

For example, suppose you are performing a printing operation and you want to control the recovery of runtime printing errors in the routine that is performing the printing operation. There are several reasons for doing this. The recovery strategy for a print error when printing labels is likely to be much different than when printing a statistical or accounting report. In the former case, rolling back the record pointer to reprint a set of labels when the printer jams is an appropriate part of the recovery strategy, but when printing reports, it is not. You may need, therefore, to define a recovery strategy for each type of printing operation. BEGIN SEQUENCE...
BREAK...END allows you to do this by branching from the error function back to your program where you can call a local recovery procedure.

To set this up, create a procedure structured like the following for each print operation where you want local recovery of print errors:

```
PROCEDURE PrintLabel
PARAMETER lbl_file
*
ok = .T.
error_state = .F.
local_err = .T.
*
DO WHILE ok
    BEGIN SEQUENCE
        LABEL FORM (lbl_file) REST TO PRINT
    END
    *
    IF error_state
        ok = Label_recover()        && Recovery routine.
        error_state = .F.
    ELSE
        ok = .F.
    ENDIF
    *
ENDDO
RETURN
```

Then create a recovery procedure as a user-defined function that returns true (.T.) if you want to continue or false (.F.) if you

want to quit the printing operation. In the recovery user-defined function, you can present a menu that allows the user to continue, quit, rollback the record pointer, view sample labels, or BREAK to a higher-level procedure.

Lastly, change your Print_error() function to include the following lines of code:

```
FUNCTION Print_error
PARAMETERS name, line
*
IF local_err
   error_state = .T.
   BREAK
ENDIF
*
<rest of error function>...
```

This example, of course, does not present an all-encompassing strategy for all situations, but it does indicate one way to use BEGIN SEQUENCE...BREAK...END to control the environment of runtime errors.

Return Method

The RETURN argument of the error function determines the action taken by Clipper when the error function terminates.

1. For some classes of errors, the error function can RETURN control to the command that instigated the error. With this type of error function, RETURNing true (.T.) forces a retry of the operation that instigated the error condition. RETURNing false (.F.) terminates the pending operation and passes control to the next intermediate code in the program.

2. For classes of errors that are more severe, RETURNing true (.T.) will retry, but RETURNing false (.F.) causes Clipper to QUIT. When this is the case, Clipper CLOSEs ALL open files.

3. For expression errors, the RETURN value is substituted for the term in which the error occurred. For example, if you attempt to evaluate the following expression containing a divide by zero error:

```
12 + (12 / 0)
```

The expression error occurs within the term (12 / 0). If you RETURN 1, the result of the expression is 13.

ERROR CLASS DESCRIPTIONS

Database Error

Database errors occur when an action is taken that affects the database file between the time it is USEd and CLOSEd. This includes operations such as APPEND BLANK or REPLACE. When this class of error occurs, the error function Db_error() is called with the following general syntax:

```
Db_error(name, line, info)
```

The "info" parameter returns one of five messages indicating the type of database error:

Database required

Occurs when you attempt to perform a database operation and one is not in USE in the work area you are accessing.

Lock required

Occurs when you attempt to operate on a file or record with an operation requiring a lock (either file or record).

Exclusive required

Occurs when you attempt a database operation that requires EXCLUSIVE USE of the database file.

Field numeric overflow

Occurs when you attempt to REPLACE a numeric field with a value that is too large to fit.

Index file corrupted

Occurs when Clipper detects damage to an index file during an update operation.

To recover, you can RETURN true (.T.) and the database operation is retried or you can branch with a BREAK to the END

of the current SEQUENCE. However, if you RETURN false (.F.), Clipper automatically executes a QUIT (CLOSEs ALL) and unceremoniously exits to DOS.

Expression Error

Expression errors can occur whenever there is an evaluation of an expression. Typically, errors of this kind are type mismatch or divide by zero errors. Whenever an error occurs during expression evaluation, the error function Expr_error() is called with the following general syntax:

```
Expr_error(name, line, info, model, _1, _2, _3)
```

The "info" parameter returns one of four messages indicating the type of expression error:

Type mismatch

Occurs when you attempt an operation where the operands are not the correct type for the specified operator.

Subscript range

Occurs when you attempt to address an array element beyond the DECLAREd dimension of the array.

Zero divide

Occurs when you attempt to divide any number by zero.

Expression error

Occurs when you attempt to expand a macro variable containing an invalid expression.

The "model" contains the expression that failed as a character string with the operand parameters in place of the expression values. The operand parameters, "_1," "_2," and "_3", themselves contain the actual operand values. For example, if you attempt to execute the following erroneous SUBSTR() expression:

```
? SUBSTR("abc", 1, "2")
```

"model" will contain the string "SUBSTR(_1,_2,_3)," "_1" will contain the character value "abc," "_2" will contain the numeric

value 1, and "_3" will contain the character string "2." To recover, you can change "_3" to a proper numeric value, evaluate "model" as a macro variable, and RETURN the result. The following demonstrates this:

```
_3 = 2
RETURN &model.
```

Another approach is to simply RETURN a value like this:

```
RETURN _1
```

Remember that the value returned by Expr_error() is taken as the result of the failed operation and execution continues from the point of the error within the expression evaluation.

Refer to the actual code in Expr_error() in Errorsys.prg for specific examples of general purpose recovery strategies.

Miscellaneous Errors

Miscellaneous errors are unclassified but non-fatal errors and occur when the error cannot be dealt with by the other error handlers. When this class of error occurs, the error function Misc_error() is called with the following general syntax:

```
Misc_error(name, line, info, model)
```

The "info" parameter contains one of two messages indicating the type of error that has occurred as follows:

Type mismatch

Occurs when you attempt to REPLACE a field with the wrong type.

RUN error

Occurs when you attempt a RUN and there is not enough memory or COMMAND.COM cannot be found. To identify the specific cause of error, use DOSERROR() to query the current DOS error.

The "model" parameter contains some fragment of Clipper code relating to the command that instigated the error. Note that unlike Expr_error(), the "model" will not contain code that can be executed as a macro.

To recover, you can retry the operation with RETURN true (.T.) or you can branch with a BREAK to the END of the current SEQUENCE. However, if you RETURN false (.F.), Clipper automatically executes a QUIT (CLOSEs ALL) and unceremoniously exits to DOS.

Open Error

Open errors occur when a file open operation fails. This includes all Clipper file operations other than those using the low-level file I/O functions. When this class of error occurs, the error function Open_error() is called with the following general syntax:

```
Open_error(name, line, info, model, _1)
```

The "info" parameter always contains the following message:

Open error

The "model" in this case contains a fragment of code indicating the operation taking place when the error occurred. The operand parameter, "_1", contains the name of the file that instigated the error.

To recover, you can RETURN true (.T.) and the open operation is retried. RETURNing false (.F.) skips the current operation and execution continues. This normally has no ill effect, except that the file in question is not in USE as expected. Note that execution may continue within the current source code line if the line contains several file open operations (i.e., SET INDEX).

Print Error

Print errors occur when any printing operation is, or becomes, not ready. This is only true, however, when you are attempting to access a parallel printer. If the printer is redirected to a serial port and the port is not ready, Open_error() is invoked instead. Note that this is independent of the device and can occur if, as an example, you are printing to a network spooler and it goes down. A print error can also occur if there is a disk error during the redirection of @...SAYs to file.

The general syntax for the print error function is as follows:

```
Print_error(name, line)
```

To recover, you can RETURN true (.T.), forcing a retry of the printer access operation. RETURNing false (.F.) skips the

printer access operation and continues execution. The latter is not recommended since you may not know what operation will execute next. The best approach is to place printing operations within a BEGIN SEQUENCE construct and BREAK to a local recovery handler if a measured number of retries fail.

Undefined Error

Undefined errors occur when you refer to a variable (memory or field) before it is defined. Typically this includes an identifier reference where TYPE() returns a "U," "UE," or "UI." When this class of error occurs, the error function Undef_error() is called with the following general syntax:

```
Undef_error(name, line, info, model, _1)
```

The "info" parameter returns one of three messages indicating the type of undefined error that has occurred:

Undefined identifier

Occurs when you specify a field, memory variable, or alias identifier not currently defined.

Not an array

Occurs when you refer to an array element and the array reference is not defined as an array.

Missing EXTERNAL

Occurs when you refer to a procedure or user-defined function and the identifier is not found. This usually happens when you have concealed a reference from both the compiler and the linker in a macro or in a LABEL or REPORT FORM.

The "model" parameter contains a code fragment you can use to indicate the operation where the error occurred. "_1" contains the undefined identifier.

To recover, you can retry the operation with RETURN true (.T.) or branch with a BREAK to the END of the current SEQUENCE. However, if you RETURN false (.F.), Clipper automatically executes a QUIT (CLOSEs ALL) and unceremoniously exits to DOS.

OTHER ERRORS

In addition to the recoverable errors described above there is a class of errors where recovery is limited to a retry of the operation that instigated the error or QUITting.

Internal error

This error generally occurs because of a bad index file. Clipper then presents the user with a message and pauses. Pressing any key QUITs.

Disk Full

Occurs when the disk is full during standard database file operations. When this happens, the user is presented with a message and the option to continue. Answering "Yes" to the prompt retries the operation; answering "No" QUITs.

Multiple Error

This error occurs when there has been an error in one of the error functions. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

Out of Memory

This error occurs when there is no more memory for the current operation to continue. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

Not Enough Memory

This error occurs when there is not enough memory to begin an operation. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

SUMMARY OF RUNTIME ERROR MESSAGES

The following is a summary of messages that either display or are passed by Clipper to error functions as the "info" parameter when a runtime error occurs. Messages that have no error function are categorized as non-recoverable. Those that have an associated error function are categorized as recoverable. If a message has an error function, the class of the error and the function name are noted.

Database required (recoverable)

Occurs when you attempt to perform a database operation and a database file is not in USE in the work area you are accessing.

Class: Database error
Function: Db_error

Disk Full (non-recoverable)

Occurs when the disk is full during standard database file operations. When this happens, the user is presented with a message and the option to continue. Answering "Yes" to the prompt retries the operation; answering "No" QUITs.

Exclusive required (recoverable)

Occurs when you attempt a database operation that requires EXCLUSIVE USE of the database file.

Class: Database error
Function: Db_error

Expression error (recoverable)

Occurs when you attempt to expand a macro variable containing an invalid expression.

Class: Expression error
Function: Expr_error

Field numeric overflow (recoverable)

Occurs when you attempt to REPLACE a numeric field with a value that is too large to fit.

Class: Database error
Function: Db_error

Index file corrupted (recoverable)

Occurs when Clipper detects damage to an index file during an update operation.

Class: Database error
Function: Db_error

Internal error (non-recoverable)

This error generally occurs because of a bad index file. Clipper then presents the user with a message and pauses. Pressing any key QUITs.

Lock required (recoverable)

Occurs when you attempt to operate on a file or record with an operation requiring a lock (either file or record).

Class: Database error
Function: Db_error

Missing EXTERNAL (recoverable)

Occurs when you refer to a procedure or user-defined function and the identifier is not found. This usually happens when you have concealed a reference from both the compiler and the linker in a macro or in a LABEL or REPORT FORM.

Class: Undefined error
Function: Undef_error

Multiple Error (non-recoverable)

This error occurs when there has been an error in one of the error functions. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

Not an array (recoverable)

Occurs when you refer to an array element and the array reference is not defined as an array.

Class: Undefined error
Function: Undef_error

Not Enough Memory (non-recoverable)

This error occurs when there is not enough memory to begin an operation. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

Open error (recoverable)

Occurs when a file open operation fails.

Class: Open error
Function: Open_error

Out of Memory (non-recoverable)

This error occurs when there is no more memory for the current operation to continue. In this case, Clipper displays a message and pauses. Pressing any key QUITs.

RUN error (recoverable)

Occurs when you attempt a RUN and there is not enough memory or COMMAND.COM cannot be found. To identify the specific cause of error, use DOSERROR() to query the current DOS error.

Class: Miscellaneous error
Function: Misc_error

Subscript range (recoverable)

Occurs when you attempt to address an array element beyond the DECLAREd dimension of the array.

Class: Expression error
Function: Expr_error

Type mismatch (recoverable)

Occurs when you attempt an operation where the operands are not the correct type for the specified operator.

Class: Expression error
Function: Expr_error

Type mismatch (recoverable)

Occurs when you attempt to REPLACE a field with the wrong type.

Class: Miscellaneous error
Function: Misc_error

Undefined identifier (recoverable)

Occurs when you specify a field, memory variable, or alias identifier not currently defined.

Class: Undefined error
Function: Undef_error

Zero divide (recoverable)

Occurs when you attempt to divide any number by zero.

Class: Expression error
Function: Expr_error

APPENDIX G

An ASCII Chart and the INKEY() Return Values

Character Displays

The chart below lists ASCII values and the character display.

Table G-1 ASCII Chart.

Val	Char	Val	Char	Val	Char	Val	Char	Val	Char	Val	Char
0		22	▬	44	,	66	B	88	X	110	n
1	☺	23	↕	45	-	67	C	89	Y	111	o
2	☻	24	↑	46	.	68	D	90	Z	112	p
3	♥	25	↓	47	/	69	E	91	[	113	q
4	♦	26	→	48	0	70	F	92	\	114	r
5	♣	27	←	49	1	71	G	93	]	115	s
6	♠	28	∟	50	2	72	H	94	^	116	t
7	•	29	↔	51	3	73	I	95	_	117	u
8	◘	30	▲	52	4	74	J	96	`	118	v
9	○	31	▼	53	5	75	K	97	a	119	w
10	◙	32		54	6	76	L	98	b	120	x
11	♂	33	!	55	7	77	M	99	c	121	y
12	♀	34	"	56	8	78	N	100	d	122	z
13	♪	35	#	57	9	79	O	101	e	123	{
14	♫	36	$	58	:	80	P	102	f	124	|
15	☼	37	%	59	;	81	Q	103	g	125	}
16	►	38	&	60	<	82	R	104	h	126	~
17	◄	39	'	61	=	83	S	105	i	127	⌂
18	↕	40	(	62	>	84	T	106	j		
19	‼	41	)	63	?	85	U	107	k		
20	¶	42	*	64	@	86	V	108	l		
21	§	43	+	65	A	87	W	109	m		

Val	Char	Val	Char	Val	Char	Val	Char	Val	Char	Val	Char
128	Ç	150	á	172	¼	194	┬	216	╪	238	∈
129	ü	151	ù	173	¡	195	├	217	┘	239	∩
130	é	152	ÿ	174	«	196	─	218	┌	240	≡
131	â	153	Ö	175	»	197	┼	219	█	241	±
132	ä	154	Ü	176	░	198	╞	220	▄	242	≥
133	à	155	¢	177	▒	199	╟	221	▌	243	≤
134	å	156	£	178	▓	200	╚	222	▐	244	⌠
135	ç	157	¥	179	│	201	╔	223	▀	245	⌡
136	ê	158	₧	180	┤	202	╩	224	α	246	÷
137	ë	159	ƒ	181	╡	203	╦	225	ß	247	≈
138	è	160	á	182	╢	204	╠	226	Γ	248	°
139	ï	161	í	183	╖	205	═	227	π	249	∙
140	î	162	ó	184	╕	206	╬	228	Σ	250	·
141	ì	163	ú	185	╣	207	╧	229	σ	251	√
142	Ä	164	ñ	186	║	208	╨	230	µ	252	ⁿ
143	Å	165	Ñ	187	╗	209	╤	231	τ	253	²
144	É	166	ª	188	╝	210	╥	232	Φ	254	■
145	æ	167	º	189	╜	211	╙	233	θ	255	
146	Æ	168	¿	190	╛	212	╘	234	Ω		
147	ô	169	⌐	191	┐	213	╒	235	δ		
148	ö	170	¬	192	└	214	╓	236	∞		
149	ò	171	½	193	┴	215	╫	237	ø		

Function Keys

Table G-2 INKEY() Return Values-Function Keys.

Function Key	Numeric Value
F1	28
F2	-1
F3	-2
F4	-3
F5	-4
F6	-5
F7	-6
F8	-7
F9	-8
F10	-9
F11 (Shift-F1)	-10
F12 (Shift-F2)	-11
F13 (Shift-F3)	-12
F14 (Shift-F4)	-13
F15 (Shift-F5)	-14
F16 (Shift-F6)	-15
F17 (Shift-F7)	-16
F18 (Shift-F8)	-17
F19 (Shift-F9)	-18
F20 (Shift-F10)	-19
F21 (Ctrl-F1)	-20
F22 (Ctrl-F2)	-21
F23 (Ctrl-F3)	-22
F24 (Ctrl-F4)	-23
F25 (Ctrl-F5)	-24
F26 (Ctrl-F6)	-25
F27 (Ctrl-F7)	-26
F28 (Ctrl-F8)	-27
F29 (Ctrl-F9)	-28
F30 (Ctrl-F10)	-29
F31 (Alt-F1)	-30
F32 (Alt-F2)	-31
F33 (Alt-F3)	-32
F34 (Alt-F4)	-33
F35 (Alt-F5)	-34
F36 (Alt-F6)	-35
F37 (Alt-F7)	-36
F38 (Alt-F8)	-37
F39 (Alt-F9)	-38
F40 (Alt-F10)	-39

Cursor Keys

Table G-3 INKEY() Return Values-Cursor Keys.

Keypad Key	Alternate	Numeric Value
Rightarrow	Ctrl-D	4
Leftarrow	Ctrl-S	19
Uparrow	Ctrl-E	5
Dnarrow	Ctrl-X	24
Ctrl-rightarrow	Ctrl-B	2
Ctrl-leftarrow	Ctrl-Z	26
Ins	Ctrl-V	22
Del	Ctrl-G	7
Home	Ctrl-A	1
PgUp	Ctrl-R	18
PgDn	Ctrl-C	3
Ctrl-Home	Ctrl-]	29
Ctrl-End	Ctrl-W	23
Ctrl-PgUp	Ctrl-hyphen	1
Ctrl-PgDn	Ctrl-ctrl	30

APPENDIX H

Reserved Words

The following words are reserved for Clipper or PLINK86 use. Words preceeded by asterisks may not be used as shown, nor in any shortened version. (Example: LIBRARY, LIBRAR, LIBRA, LIBR, LIB, LI and L are all reserved words.) Although most function names can be redefined, it is suggested that they not be used as procedure or user-defined function names. Finally, do not use underscores (_) within procedure, format, function or program names.

Table H-1 Reserved Words

*	BATCH	*	NOBELL
*	BEGINAREA	*	OUTPUT
	CODE		PROCFILE
	DATA		READEXIT
*	DEBUG		READINSERT
*	ENDAREA	*	SEARCH
	ERRORLEVEL	*	SECTION
	INDEXEXT		SETPRC
	INDEXORD		SUMMER87
*	FILE		SYSTEM
*	HEIGHT	*	UPPERCASE
*	LIBRARY	*	VERBOSE
*	LOWERCASE	*	WIDTH
*	MAP	*	WORKFILE
	NETERR		

APPENDIX I

PLINK86-Plus Commands

#

The comment marker, consistent with the syntax used by PMAKER.

BATCH

Causes PLINK86-Plus to terminate with a fatal error if it cannot find a specified .OBJ or .LIB file. (By default, the operator is prompted when a specific file cannot be found.)

BEGINAREA

[Abbreviated "BEGIN"]. Defines the start of an overlay area. Each overlay area must begin with a BEGINAREA statement and end with an ENDAREA statement.

DEBUG

Provides information to aid in debugging. Displays the name of the overlay as it is loaded during program execution.

ENDAREA

[Abbreviated "END"]. Defines the end of an overlay area. Each overlay area must begin with a BEGINAREA statement and end with an ENDAREA statement.

FILE

[Abbreviated "FI"]. Identifies an object module file. The DOS filename extension (.OBJ) need not be specified. Several object modules may be identified with a single FILE command; separate the module names with a comma.

HEIGHT

Changes the page height of the memory map reports to the specified number of lines when HEIGHT is followed by that number. The default is 65 lines per page.

LIBRARY

[Abbreviated "LIB"]. Identifies the runtime libraries required to be linked to your object modules. .LIB is the default file type unless otherwise specified.

PLINK86-Plus will automatically look for the CLIPPER.LIB file. If you are using overlays, PLINK86-Plus will also look for the OVERLAY.LIB file. You need not use this command if you will be using only those two libraries.

LOWERCASE

Translates all identifiers and symbols to lower case.

MAP

Produces memory map reports that describe the output of the linkage edit including memory usage, size and location information. MAP is useful for assembly language debugging, for memory management and planning, and for linkage edit testing and debugging.

Enter the MAP command as follows:

MAP = <filename> [A] [, G] [, M] [, S]

PLINK86-Plus will assign a .MAP filename extension to the filename you specify. The different reports are specified as follows:

A - All (Segments). Lists program sections in input order with segments of each section listed in order of ascending memory address.

G - Global (Public) Symbols. Lists the public symbols in alphabetical order with the memory addresses for each.

M - Modules. Lists each program module in input order with its memory address and size. Each segment contained in the module, and each symbol contained in the segment is also listed with their addresses and size. This report provides the most detailed information.

S - Sections. Lists all program sections alphabetically with summary information for each.

You may specify any or all of the above reports. If none are specified, the "A" report will be produced.

NOBELL

Suppresses the beeps associated with the PLINK86-Plus messages.

OUTPUT

Identifies the name of the load module file. If this command is not specified, the load module filename will become the name of the first filename identified by a FILE command. You need not specify the DOS filename extension; it will automatically be assigned. OVERLAY specifies the names of the segment classes that will be allowed to remain in the overlay structure.

SEARCH

Similar to the LIBRARY statement but causes PLINK86-Plus to make multiple passes through the specified library files if undefined symbols remain after the files have been read. .LIB is the default file type unless otherwise specified.

SECTION

Identifies the following object modules as those that are to be contained in an internal overlay.

SECTION INTO

Identifies the following object modules as those that are to be contained in an external (.OVL) file.

UPPERCASE

Translates all identifiers and symbols to upper case.

VERBOSE

Displays the current operations of the linker on the last line of the screen. Note that this statement slows down the linkage edit considerably. Do not use this command when you are using the DOS output redirection feature.

WIDTH

Changes the page width of the memory map reports to the specified number of characters per line when WIDTH is followed by that number. The default is 80 characters per line.

WORKFILE

Redirects the PLINK86-Plus interim disk file, if required.

APPENDIX J

Sample Programs

This appendix presents a number of user-defined functions and programs that serve both as examples and workarounds for common programming problems. The examples are divided into sections based on the category of usage as follows:

- dBASE III PLUS Compatibility

Dbf()	Name of current alias.
FkLabel()	Function key name.
FkMax()	Number of function keys constant.
Mod()	Modulus of two numbers.
Os()	Operating system name constant.
ReadKey()	Similar to dBASE III PLUS READKEY().
Version()	Clipper version constant.

- User Interface

Dup_Chk()	Duplicate entry check.
Multi_Form()	Multiple-page format screens.
ValidTime()	Valid time string entry check.
AcceptAt()	Keyboard input at a screen location.

- Low Level File I/O

Philes.prg	Copy a file.

- String Functions

Currency()	Numeric to dollar format.
StrZero()	Numeric to string with leading zeros.

- Time Functions

AmPm()	Time string to 12-hour format.
Days()	Numeric seconds to days.
ElapTime()	Difference between timestrings.
Secs()	Seconds from midnight given a time string.
Tstring()	Numeric seconds to time string.

- Numeric Function
  LenNum() — Length of numeric including decimals.

- Arrays
  Dim2() — Array addressing with 2-dimensions.

dBASE III PLUS COMPATIBILITY

The following user-defined functions provide added compatibility with dBASE III PLUS. All functions in this section are provided on disk in the file Examplep.prg and already linked into EXTEND.LIB.

Dbf()

Dbf() returns the name of the alias of the current work area using ALIAS() with no argument. This is not identical to dBASE III PLUS, since it does not return the drive and the extension of the current database file. If there is no active database file in the specified work area, DBF() returns a null ("") string.

FUNCTION Dbf
Syntax: string = Dbf()
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Dbf
RETURN ALIAS()
```

FkLabel()

FkLabel() simulates the dBASE III PLUS FKLABEL() function by returning a character string in the form "F1" depending on the numeric value you pass to it. It is intended to return the name of the <expN>th programmable function key. If the numeric argument is less than one or greater than 40, it returns a null ("") string.

FUNCTION FkLabel
Syntax: string = FkLabel(<expN>)
Author: Tom Rettig
Date : 11/01/85, 09/01/86

```
FUNCTION Fklabel
PARAMETERS cl_1
RETURN IF(cl_1 <= 40 .AND. cl_1 > 0, "F" +;
   LTRIM(STR(cl_1)), [])
```

FkMax()

FkMax() returns the maximum number of programmable function keys on the computer. In this case it returns a constant, 40.

FUNCTION FkMax
Syntax :n = FkMax()
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Fkmax
RETURN 40     && IBM specific
```

Mod()

Mod() emulates the dBASE III PLUS MOD() function using the Clipper modulus operator (%). Note, however, there are differences between the dBASE III PLUS MOD() function and the Clipper modulus operator as indicated in the following table:

Table J-1 Differences Between dBASE III PLUS MOD() Function and the Clipper Modulus Operator

Clipper Operator	dBASE III PLUS Function
3 % 0 ::= 0.00	MOD(3, 0) ::= 3
3 % -2 ::= 1.00	MOD(3,-2) ::= -1
-3 % 2 ::= -1.00	MOD(-3, 2) ::= 1
-3 % 0 ::= 0.00	MOD(-3, 0) ::= -3
-1 % 3 ::= -1.00	MOD(-1, 3) ::= 2
-2 % 3 ::= -2.00	MOD(-2, 3) ::= 1
2 % -3 ::= 2.00	MOD(2,-3) ::= -1
1 % -3 ::= 1.00	MOD(1,-3) ::= -2

FUNCTION Mod
Syntax: n = Mod(<expN1>, <expN2>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Mod
PARAMETERS cl_num, cl_base
PRIVATE cl_result
cl_result = cl_num % cl_base
RETURN IF(cl_base = 0, cl_num,;
    IF(cl_result * cl_base<0, cl_result + cl_base, ;
    cl_result))
```

Os()

Os() returns a constant in order to emulate the dBASE III PLUS equivalent.

FUNCTION Os
Syntax: string = Os()
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Os
RETURN "MS/PC-DOS"  && DOS specific, MicroSoft or IBM
```

ReadKey()

ReadKey() returns an integer numeric value to return a code representing the key pressed to complete a full-screen operation. The following keys are supported:

Table J-2 ReadKey() Return Codes

Exit Key	Return Code
PgUp	6
PgDn	7
Esc	12
Ctrl-End, Ctrl-W	14
Type past end	15
Return	15

If UPDATED() is true (.T.), ReadKey() returns the code plus 256.

FUNCTION ReadKey
Syntax: n = ReadKey()
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Readkey
DO CASE
CASE LASTKEY() = 18            && PgUp
   RETURN  6 + IF(UPDATED(), 256, 0)
CASE LASTKEY() =  3            && PgDn
   RETURN  7 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 27            && Esc
   RETURN 12 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 23            && Ctrl-W
   RETURN 14 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 13            && Enter
   RETURN 15 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 31            && Ctrl-PgUp
   RETURN 34 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 30            && Ctrl-PgDn
   RETURN 35 + IF(UPDATED(), 256, 0)
CASE LASTKEY() = 32            && type past end
   RETURN 15 + IF(UPDATED(), 256, 0)
ENDCASE
```

Version()

Version() returns the name of the current version of Clipper as a constant. Depending on how you organize your libraries, you may want to change this whenever you update your Clipper

version. Version() does, however, indicate the version of Clipper EXTEND.LIB is associated with.

FUNCTION Version
Syntax: string = Version()

```
FUNCTION Version
RETURN "Clipper, Summer '87"
```

USER INTERFACE

The following are user-defined functions that demonstrate various user-interface applications.

Dup_Chk()

Dup_Chk() is a validation function you can use with the VALID clause of @...GET. Its primary function is to perform a lookup in a specified work area and return an error if a matching record is found. Prior to performing the lookup check, two other checks are made. First, if the GET variable is blank, Dup_Chk() returns true (.T.). Second, if the GET variable is not completely filled, an error message displays and Dup_Chk() returns false (.F.).

To call Dup_Chk(), you must specify both the name of the current GET variable and the lookup work area .

FUNCTION Dup_Chk
Syntax.: logical = Dup_Chk(<variable>, <work area>)
Author: Ray Love
Date: June 1, 1986 modified September 1, 1987

```
FUNCTION Dup_Chk
PARAMETERS dup_no, workarea

* An empty value is acceptable.
IF EMPTY(dup_no)
   RETURN .T.
ENDIF

* Integrity check.
IF LEN(TRIM(dup_no)) < LEN(dup_no)
   Err_msg("Field not completely filled")
   RETURN .F.
ENDIF

* Duplicate check.
lastarea = SELECT()
SELECT workarea
SEEK dup_no
IF FOUND()
```

```
    Err_msg("Already on file")
    validation = .F.
ELSE
    validation = .T.
ENDIF
SELECT (lastarea)
RETURN validation

FUNCTION Err_msg
PARAMETERS msg
SAVE SCREEN
row = 7
msg = msg + ", press any key..."
col = INT((80 - LEN(msg))/2) - 2
@ row, col CLEAR TO row + 2, col + LEN(msg) + 4
@ row, col TO row + 2, col + LEN(msg) + 4 DOUBLE
@ row + 1, col + 2 SAY msg
INKEY(0)
RESTORE SCREEN
RETURN ""
```

Multi_Form()

Multi_Form() is a general purpose format paging routine. It can, however, be used to emulate multiple-screen format files supported by dBASE III PLUS. To use Multi_Form(), first create an array filling it with the names of the format procedures in the order you want them executed. Then when you call Multi_Form(), specify the array as the sole argument. For example:

```
DECLARE arry[2]
arry[1] = "Customers"
arry[2] = "Invoices"
lastpage = Multi_Form(arry)
```

Multi_Form() pages forward between format screens when the last GET in a format terminates. PgUp and PgDn move forward or backward one screen. Esc terminates Multi_Form() returning the position in the array of the last format screen processed.

FUNCTION Multi_Form
Syntax: n = Multi_Form(<arrayC>)
Author: Ira Emus
Date: December 3, 1986, modified September 1, 1987
Note(s): Paging mechanism for screen formats.

```
FUNCTION Multi_Form
PARAMETERS scr_arry
PRIVATE pg_count, form
*
maxpage  = LEN(scr_arry)
pg_count = 1
DO WHILE .T.
    CLEAR
    form = scr_arry[pg_count]
    SET FORMAT TO &form.
    READ
    DO CASE
    CASE LASTKEY() = 27
        *
        * Esc key.
        EXIT
    CASE LASTKEY() = 18
        *
        * PgUp key.
        IF pg_count > 1
            pg_count = pg_count - 1
        ELSE
            EXIT
        ENDIF
    OTHERWISE
        *
        * Any other key.
        IF pg_count < maxpage
            pg_count = pg_count + 1
        ELSE
            EXIT
        ENDIF
    ENDCASE
ENDDO
RETURN pg_count
```

ValidTime()

ValidTime() is used with the VALID clause of @...GET to validate a time string. For example:

```
timestring = [  :  :  ]
@ 10, 10 GET timestring PICTURE [99:99:99];
    VALID ValidTime(timestring)
```

A valid time string comprises eight characters in the form HH:MM:SS with the range 00:00:00 to 23:59:59.

FUNCTION ValidTime
Syntax: logical = ValidTime(<time string>)

Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION ValidTime
PARAMETERS timestring
RETURN VAL(timestring)          < 24 .AND.;
    VAL(SUBSTR(timestring, 4)) < 60 .AND.;
    VAL(SUBSTR(timestring, 7)) < 60
```

AcceptAt()

AcceptAt() emulates the ACCEPT command allowing you to input printable characters beginning at a specified screen location. This capability is primarily used to GET keyboard entry within SET KEY procedures activated during READs. Clipper does not support nested READs prohibiting you from GETting keyboard entry from a specified screen location and ACCEPT is not particularly useful since it performs a carriage return/line feed before performing its business.

To call AcceptAt(), specify the row and column coordinate, and the prompt string as arguments. For example:

```
var = AcceptAt(12, 12, "Enter: ")
? var
```

AcceptAt() then accepts keyboard input until Return is pressed returning keyboard input as a character string.

FUNCTION AcceptAt
Syntax: string = AcceptAt(<row>, <col>, <prompt>)
Author(s):Ray Love and Fred Ho
Date: December 1, 1986 modified September 1, 1987

```
FUNCTION AcceptAt
PARAMETERS row, col, prompt
PRIVATE char, string
*
STORE "" TO char, string
@ row, col SAY prompt
DO WHILE .T.
    char = INKEY(0)
    DO CASE
    CASE char = 13
        *
        * Return key.
        EXIT
    CASE char > 31 .AND. char < 127
        *
        * Printable characters.
```

```
        string = string + CHR(char)
        @ ROW(), COL() SAY CHR(char)
    CASE (char = 8 .OR. char = 19) .AND. LEN(string) > 0
        *
        * Backspace or Leftarrow keys.
        @ ROW(), COL() - 1 SAY " "
        @ ROW(), COL() - 1 SAY ""
        string = SUBSTR(string, 1, LEN(string) - 1)
    ENDCASE
ENDDO
RETURN string
```

LOW LEVEL FILE I/O

The following program, Philes.prg, demonstrates typically what is required to program using the low level file functions.

```
*  Philes.prg
? "Copy a file..."

size = 512
buffer = SPACE(size)
total  = 0
remaining = 0

ACCEPT "Source file:  " TO source
ACCEPT "Target file:  " TO target
?

shandle = FOPEN(source)
IF FERROR() = 0

    thandle = FCREATE(target)
    IF FERROR() = 0

        * Get total source file size.
        total = FSEEK(shandle, 0, 2)
        remaining = total

        * Reset file position.
        FSEEK(shandle, 0)

        DO WHILE (remaining > 0)

            IF (remaining < size)
                size = remaining    && Last part of file.
            ENDIF

            scount = FREAD(shandle, @buffer, size)
            IF scount <> size
                ? "Error reading", source
                EXIT
```

```
            ENDIF

            tcount = FWRITE(thandle, buffer, size)
            IF tcount <> size
               ? "Error writing", target
               EXIT
            ENDIF

            remaining = remaining - size

         ENDDO

         FCLOSE(thandle)

      ELSE
         ? "Cannot create", target, ", DOS error", FERROR()
      ENDIF

      FCLOSE(shandle)

   ELSE
      ? "Cannot open", source, ", DOS error", FERROR()
   ENDIF

   ? "Total of", total - remaining, "bytes transfered."
   RETURN
```

STRING FUNCTIONS

The following two user-defined functions format numerics converting them into character strings.

Currency()

Currency() converts any numeric argument having up to 12 whole number digits into a character string formatted as money. This format has embedded commas, two decimal places, and a floating dollar sign.

FUNCTION Currency
Syntax: string = Currency(<expN>)
Date: September 1, 1987

```
FUNCTION Currency
PARAMETERS p1
PRIVATE outstr
outstr = LTRIM(TRANSFORM(p1, "999,999,999,999.99"))
RETURN(SPACE(LEN(STR(p1)) - LEN(outstr) + 1) + ;
     "$" +  outstr)
```

StrZero()

StrZero() returns the STR() of a numeric expression with leading zeros instead of blanks.

FUNCTION StrZero
Syntax :string = StrZero(<expN1> [,<expN2> [,<expN3>]])
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION StrZero
PARAMETERS cl_num, cl_len, cl_dec
PRIVATE cl_str
*
DO CASE
CASE PCOUNT() = 3
   cl_str = STR(cl_num, cl_len, cl_dec)
CASE PCOUNT() = 2
   cl_str = STR(cl_num,cl_len)
OTHERWISE
   cl_str = STR(cl_num)
ENDCASE

IF "-" $ cl_str
   * Negative number, move the minus sign in front of
   * zeros.
   RETURN "-" + REPLICATE("0", LEN(cl_str) - ;
   LEN(LTRIM(cl_str))) + SUBSTR( cl_str, AT("-",
cl_str) + 1)
ELSE
   * Positive number.
   RETURN REPLICATE("0", LEN(cl_str) - ;
   LEN(LTRIM(cl_str))) + LTRIM(cl_str)
ENDIF
```

TIME FUNCTIONS

The following user-defined functions present an integrated system of routines for time manipulation. These include both calculation and formatting.

AmPm()

AmPm() takes a time string in the format HH:MM:SS and returns an 11-byte string in 12-hour am/pm format.

FUNCTION AmPm
Syntax : string = AmPm(<time string>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION AmPm
PARAMETERS cl_time
RETURN IF( VAL(cl_time) < 12, cl_time + " am",;
   IF( VAL(cl_time) = 12, cl_time + " pm",;
      STR(VAL(cl_time) - 12, 2) + SUBSTR(cl_time, 3) +
       " pm" ))
```

Days()

Days() returns the number of days from numeric seconds to the nearest day. The remainder under 24 hours can be obtained by Tstring(<seconds> % 86400)

FUNCTION Days
Syntax :n = Days(<expN>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Days
PARAMETERS cl_secs
RETURN INT(cl_secs / 86400)
```

ElapTime()

ElapTime() calculates the difference between two time strings in the format HH:MM:SS returning the result as a time string. This should only be used for timings under 24 hours.

Note that ElapTime() is dependent on the user-defined functions, Secs() and Tstring(), also in this appendix.

FUNCTION ElapTime
Syntax: timestring = ElapTime(<expC1>, <expC2>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION ElapTime
PARAMETERS cl_start, cl_end
RETURN Tstring(IF(cl_end < cl_start, 86400 , 0) +;
   Secs(cl_end) - Secs(cl_start))
```

Secs()

Secs() returns the number of seconds from midnight of a time string in the form HH:MM:SS. The maximum return value is 86,400, the numbers of seconds in a day. The inverse of Secs() is Tstring().

FUNCTION Secs
Syntax: n = Secs(<expC>)
Author: Tom Rettig
Date : 11/01/85, 09/01/86

```
FUNCTION Secs
PARAMETERS cl_time
RETURN VAL(       cl_time   ) * 3600 +;
   VAL(SUBSTR(cl_time,4)) *   60 +;
   VAL(SUBSTR(cl_time,7))
```

Tstring()

Tstring() converts numeric seconds to a 24-hour time string. The maximum number of seconds that be converted is 86,400 (the number of seconds in one day). Time quantities over 24 hours are returned by Days().

The inverse of Tstring() is Secs().

Note that Tstring() is dependent on the user-defined functions StrZero() and Mod() also in this appendix.

FUNCTION Tstring
Syntax: time string = Tstring(<expN>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION Tstring
PARAMETERS cl_secs
RETURN StrZero(INT(Mod(cl_secs/3600, 24)), 2, 0);
   + ":" +;
   StrZero(INT(Mod(cl_secs/  60, 60)), 2, 0) ;
   + ":" +;
   StrZero(INT(Mod(cl_secs     , 60)), 2, 0)
```

NUMERIC FUNCTION

LenNum()

LenNum() returns the number of digits that result from a numeric expression including the decimal point and any decimal digits. LenNum() uses STR() to convert the numeric value to a character string in order to determine the length. Refer to the entry for STR() in Chapter 5 for the rules used to convert numeric values to character strings.

FUNCTION LenNum
Syntax: n = LenNum(<expN>)
Author: Tom Rettig
Date: 11/01/85, 09/01/86

```
FUNCTION LenNum
PARAMETERS cl_number
RETURN LEN(LTRIM(STR(cl_number)))
```

ARRAYS

Dim2()

Clipper only supports one-dimensional arrays. You can, however, create two-dimensional arrays using the following user-defined function, Dim2(). This user-defined function provides the position-allocator mechanism to access an element based on a row and column coordinate. For example:

```
rows = 5
cols = 2
*
DECLARE array[rows * cols]
*
array[Dim2(2, 1)] = "A single value"
*
? array[Dim2(2, 1)]
```

Note that for this to work, the memory variable, "rows" must exist and contain the number of rows defined in the two-dimensional matrix.

FUNCTION Dim2
Syntax: n = Dim2(<expN1>, <expN2>)
Author: Fred Ho
Date: September 9, 1986

```
FUNCTION Dim2
PARAMETERS x, y
RETURN (((x - 1) * rows) + y)
```

GLOSSARY

Algorithm The description of steps required to perform a task.

Alias The name of a work area. An alternate name given to a database file. Aliases are often used to give database files descriptive names and are assigned when the database file is opened. If no alias is specified when the database file is USEd, the name of the database file becomes the alias.

Application A program system designed to execute a class of interrelated tasks.

Argument A value or variable passed to a function or procedure.

Array A data structure of fixed length whose constituent elements can be referred to by position.

Assignment A statement that copies a value to a variable. In Clipper this is done with the assignment operator (=) or the STORE command.

Attribute

1. With screen color it refers to intensity, inverse video, and underlining.

2. As a formal DBMS term, it describes a column in a table or a field in a database file.

Beginning-of-file The top of the database file. In Clipper there is no beginning-of-file area or record. Instead, it is indicated by BOF() returning true (.T.) if an attempt is made to move the record pointer above the first record in the database file or the database file is empty.

Buffer A temporary data storage location in memory.

Clause An optional or required section of a Clipper command beginning with a keyword that modifies or enhances the command.

Column A numeric expression that evaluates to an integer identifying a screen or printer column position.

Comment A whole or part of a line of code ignored by the compiler.

Concatenate To combine two groups of character data together by placing them in a sequence.

Condition A logical expression used within the WHILE or FOR clauses for record processing commands to define a set of records to process.

Constant A constant refers to the representation of an actual value. For example, .T. is a logical constant, "string" is a character constant, 21 is a numeric constant. There are no date and memo constants.

Control Structure Any program structure that alters the flow of program control. In Clipper, these include DO WHILE...ENDDO, DO CASE...ENDCASE, FOR...NEXT, and IF...ELSE...ENDIF.

Controlling/ Master Index The index currently being used to refer to records by key value or sequential record movement commands.

Crash Master A member of a software development team who can find undefined boundary conditions and fundamental flaws in a program's design seemingly at will.

Data Type In Clipper, there are five data types: character, numeric, date, logical, and memo. Fields can be any type. Memory variables can be any type but memo. All valid character operations, however, can be performed on memo fields.

Database An aggregation of related operational data used by application systems. A database can contain one or more data files.

DBMS An acronym for the term "database management system." A DBMS is a software system that mediates access to the database through a data manipulation language.

Decrement To decrease a value by a fixed amount, usually one.

Delimiter A symbol that marks a boundary. In Clipper, this term usually applies to delimited files where it specifies the character that bounds character fields. Note that it is the field separator which is in fact the comma.

Destination The work area or device to which data is sent.

Drive	A letter designating a disk drive.
Element	The component unit of an array and usually pointed to by a numeric subscript or index.
End-of-file	The bottom of a database file. In Clipper, this is LASTREC() + 1 and indicated by EOF() returning true (.T.).
Escape key	Any key used to terminate a mode.
Evaluate	The process of executing an expression and returning a result.
ExpC	An expression that evaluates to character type.
ExpD	An expression that evaluates to date type.
ExpL	An expression that evaluates to logical type. It is also referred to as a "condition."
ExpN	An expression that evaluates to numeric type.
Expression	Any combination of characters, operators, functions, and identifiers that can be evaluated.
Ext	A filename extension. This term is used only in the command syntax where file extensions are required.
Field	The basic column unit of a database file. A field has four attributes: name, type, length, and decimals if the type is numeric.
Filename	The name of a disk file that may or may not include a drive designator, path, and extension. If a command or function requires an extension, the "ext" term is specified in the syntax reference.
Function	In Clipper, there are two types of functions, internal and user-defined. User-defined functions begin with the keyword FUNCTION and contain a RETURN with an argument.
Identifier	A user determined name supplied to the compiler identifying a memory variable, field, or procedure. Identifiers must begin with an alphabetic character and only the first ten characters are significant.

Increment — To increase a value by a fixed amount, usually one.

Index — A set of key values organized and maintained in a tree structure, each key pointing to a record in the related database file. In Clipper indexes are maintained within special (.ntx or .ndx) files, one index per file.

Initialize — To give a memory variable a starting value. In Clipper, the type of the initializing value establishes the data type of the memory variable.

Integer — A number with no decimal digits. Note that in Clipper there is no integer type.

Iteration — One of the three basic building blocks of algorithm development (the others are sequence and selection). Iteration is a repetitive operation, that like selection, allows control to flow according to circumstance.

Join — An operation that takes two tables as operands and produces one table as a result. It is in fact a combination of other operations including selection and projection.

There are several different join operations: equijoin, greater-than join, and natural join.

Key expression — An expression used to INDEX or TOTAL a database file.

Kludge — An inelegant solution devised to overcome a basic limitation in a software system.

List — An enumeration of items separated by commas. Items can be memory variables, fields, expressions, or filenames.

Macro — A memory variable identifier that is literally substituted by its contents. In Clipper, macros are valid in expressions or wherever literal identifiers are required.

Memvar — A memory variable name.

Mnemonic — A memory aid. In computing, a mnemonic is an identifier that gives a logical name to an object.

Module

Self-contained procedures (algorithms) that are generally data independent and used to compartmentalize common functions for use throughout a system. This is accomplished principally by passing parameters.

In Clipper, you can use procedures and user-defined functions to create modular programs.

Natural order

The order records are entered into the database file. This is the same as the database file in unindexed order.

Normalization

Normalization is the process of elimination and consolidation of redundant data elements in a database.

Operand

A term that exists on either side of an operator.

Operator

Symbols identifying operations to be performed on data. Operators in Clipper fall into four classes: logical, mathematical, relational, and string. In addition, there are two types of operators: unary and binary. Unary requires only one operand where binary requires two.

Parameter

Parameters are defined as either formal or actual. Formal parameters are the receiving memory variables specified as arguments of the PARAMETERS command. Actual parameters are the arguments of the calling DO...WITH or the user-defined function.

There are two methods of passing parameters: by value or by reference. By value means that the actual parameter is evaluated and the result is placed in a memory location. When the subsequent PARAMETERS command executes, the value is transferred to the receiving variable. Passing by reference, by contrast, means that a pointer to the location of the actual parameter is passed instead of the value. Subsequent changes to the formal parameter are actually changes to the actual parameter, hence the term passing by reference.

Note that in Clipper there is no argument checking and therefore no requirement that the number of actual parameters match the number of formal parameters.

Path

A pointer to a directory. It includes a list of all the directories from the root to the specified directory separated by backslashes.

A path list then is the sequence of paths to search, each separated by a comma or semi-colon.

Procedure

Any executable block of code beginning with the statement PROCEDURE <procedure name>.

Procedure file

A file containing procedures and user-defined functions.

Projection

A DBMS term specifying a subset of fields. In Clipper, the analogy is the FIELDS clause.

Prompt

A series of characters displayed on the screen indicating that input from the keyboard is expected.

Queue

A data structure of variable length where elements are added to one end and retrieved from the other. A queue is often described as "first in, first out."

Record

The basic row unit of a database file consisting of one or more field elements.

Recursion

The process of a procedure or function calling itself.

Reference

A method by which memory variables are passed to procedures and user-defined functions. Passing by reference is a local renaming of a global variable. Changes made to a new variable name are reflected when the previous variable name is accessed.

See also: parameter.

Row

A numeric expression that evaluates to an integer identifying a screen or printer row position.

Runtime error

An error that halts a program while it is executing.

Scope

1. A command clause that specifies a range of database records to be addressed by the command. The scope clause uses the qualifiers ALL, NEXT, RECORD, and REST to define the record scope.

2. The part of the program where a memory variable is defined. In Clipper there are two classes of memory variable scope: public and private. The scope of a public memory variable is all procedures and user-defined functions in a program system. The scope of a private memory variable is the procedure in which it is created and all procedures and user-defined functions below it. This means that when the procedure where a memory variable is created terminates, the memory variable is released.

Moreover, all memory variables are private unless explicitly declared PUBLIC. Declaring memory variables PRIVATE, by contrast, hides public and inherited private memory variables created in other procedures and creates a new variable whose scope is the current procedure.

Scoreboard

An area on line 0 beginning at column 60 that displays the message "<insert>" when the insert mode is on during READ or MEMOEDIT(). It is also used for the RANGE error message if the RANGE clause is specified for a GET. This color display of the scoreboard area is the standard setting (the same as SAYs).

Selection

1. One of the three basic building blocks of algorithm development (the others are sequence and iteration). Selection allows control to flow along a number of possible paths, the actual path depending on the circumstance encountered.

2. A DBMS term that specifies a subset of records meeting a condition. The selection itself is obtained with a selection operator. In Clipper, the analogy is the FOR clause.

Separator

The character or set of characters that differentiate fields or records from one another.

See also: delimiter.

Sequence

Sequence is one of the three basic building blocks of algorithm development (the others are selection and iteration) and has two basic properties: discreteness and order. Within an algorithm there are first, a series of discrete steps necessary to accomplish a goal and second, certain steps must be performed before others, thereby imposing order. As an example, you must first USE a database file before you can access its fields.

Skeleton

A wildcard mask used to specify a group of filenames or memory variables. The "*" is used to specify one or more characters and "?" to specify a single character.

Stack

A data structure of variable length whose elements are added and retrieved from the same end. A stack is often described as "first in, last out."

String

A series of characters including letters, numbers, and symbols.

Stub

A procedure used for debugging purposes that only simulates the intended actions of the real procedure. It may display an indicating message, return a constant value, or do nothing.

Subscript

A numeric expression whose result points to a specific element position in an array. It is also referred to as an index.

Tuple

A formal DBMS term that refers to a row in a table or a record in a database file. In DIF files, tuple also refers to the equivalent of a record.

Unary

Refers to an operator having a single operand. A typical example is the .NOT. operator that has the syntax: .NOT. <condition>.

Undefined

A memory variable that exists but has not been assigned a value.

User Function

A user-defined function called by ACHOICE() or DBEDIT() to handle key exceptions. User functions are passed to either of these functions as a character string.

User-defined function

Any executable block of code beginning with the statement FUNCTION <name> and containing a RETURN statement with an argument. In Clipper, all user-defined functions must return a value although internal functions do not.

Value

A method by which memory variables are passed to procedures and user-defined functions. Passing by value means that an actual parameter is evaluated and the result is placed in a memory location. When the subsequent PARAMETERS command executes, the value is transferred to the receiving variable.

See also: reference, parameter.

Variable

An identifier holding a value that can evaluated. In Clipper this is a memory variable or database field.

Vector

In a DIF file, vector refers to the equivalent of a Clipper field.

View

A DBMS term that defines a virtual table. A virtual table does not actually exist but is derived from existing tables and maintained as a definition. The definition in turn is maintained in a separate file or as an entry in a system dictionary file.

Wait State

A mode that fetches keys from the keyboard buffer and also executes SET KEY procedures. These include ACCEPT, INPUT, MENU TO, READ, and WAIT.

There are, however, a number of functions that fetch keys from the keyboard buffer but are not considered wait states since they do not execute SET KEY procedures. These include ACHOICE(), DBEDIT(), and INKEY().

Work Area

The basic containment area of a database file and its associated indexes. Work areas can be referred to by alias name, number, or a letter designator.

Index

! (.NOT.) 4-11
!/RUN 5-12
!= (Not Equal to) 4-11
(Not Equal to) 4-11
(Comment, MAKE) 12-15
(Comment, PLINK86-Plus) I-1
$ (Subset of) 4-11
AT() 6-31
RAT() 6-108
$ (Macro, MAKE) 12-15
.$$$ File (Temporary File) 7-5
$* (Special Macro, MAKE) 12-17
$** (Special Macro, MAKE) 12-19
$@ (Special Macro, MAKE) 12-19
% (Modulus) 4-10
% (Comment, PLINK86 -older versions) 7-14
& (Macro Substitution) 4-17
See also Macro
&& (Clipper Comment) 5-11, 5-94
() (Change order of operations) 4-11
* (Multiplication) 4-10
* (Clipper Comment) 5-11, 5-94
** (Exponentiation) 4-10
+ (Addition) 4-10, 4-12, 4-15
+ (Concatenate) 4-15
... (Ellipsis) 4-16
/ (Division) 4-10
/ (Slash, either/or) 4-16
/SEGMENTS (DOS Link Option) 7-8
; (Continue Command Line) 4-15
; (Path List Separator) 5-157
= (Equal to) 4-11
== (Comparison) 4-11
?/?? (Display Result) 5-5, 5-20
[ ] (Square brackets) 4-16
^ (Exponentiation) 4-10
8087, 80287, 80387 math co-processor chip 2-2, 4-24
@...BOX 3-3, 3-6, 3-10, 5-5, 5-21
@...CLEAR 4-21, 5-5, 5-23
SCROLL() 6-115
@...GET 10-6
@...PROMPT 3-5, 3-10, 5-5 5-24
SET MESSAGE 5-155
@...SAY...GET 3-10, 4-4, 5-5, 5-25 - 5-29, 10-13
@...TO 5-5, 5-30
__PARC 11-3, 11-32 - 11-33, 11-45
__PARCLEN 11-3, 11-33
__PARCSIZ 11-3, 11-33 - 11-34
__PARDS 11-3, 11-34 - 11-35, 11-45
__PARINFA 11-3, 11-35 - 11-36
AX Values 11-36
__PARINFO 11-3, 11-36 - 11-37
AX Values 11-37
__PARL 11-3, 11-37 - 11-38
__PARND 11-3, 11-38 - 11-39, 11-45
__PARNI 11-3, 11-39
__PARNL 11-3, 11-39 - 11-40, 11-45
__RET 11-3, 11-40 - 11-41
__RETC 11-3, 11-40
__RETCLEN 11-3, 11-41

__RETDS 11-3, 11-41 - 11-42
__RETL 11-3, 11-42
__RETND 11-3, 11-42 - 11-43
__RETNI 11-3, 11-43
__RETNL 11-3, 11-43
_exmback() 11-7
_exmgrab() 11-7 - 11-8
_parc() 11-3, 11-8
_parclen() 11-3, 11-41
_parcsiz() 11-3, 11-8 - 11-9
_pards() 11-3, 11-10 - 11-11
_parinfa() 11-3, 11-11 - 11-13
_parinfo() 11-3, 11-13 - 11-15
_parl() 11-3, 11-15
_parnd() 11-3, 11-15 - 11-16
_parni() 11-3 , 11-16
_parnl() 11-3, 11-16
_ret() 11-3, 11-16 - 11-17
_retc() 11-3, 11-17
_retclen() 11-3, 11-17 - 11-18
_retds() 11-3, 11-18
_retl() 11-3, 11-18
_retnd() 11-3, 11-18 - 11-19
_retni() 11-3, 11-19
_retnl() 11-3, 11-19

A

ABS() 6-3, 6-14
ACCEPT 4-21, 5-5, 5-31, D-1
    LASTKEY() 6-82
    SET CONSOLE 5-132
Accept_At() J-9
Access, DOS 5-118
ACHOICE() 3-5, 5-86, 6-3, 6-15 - 6-18, 6-24
    Active Keys - with Function 6-17
    Active Keys without Function 6-16
    Modes 6-17
    Return Values 6-18
ACOPY() 6-3, 6-19
Active Keys, DBEDIT() 6-43
Active Memory Variables 3-2, 4-8
Add Records
    See APPEND
        BLANK, ADD_REC()
ADD_REC() 10-13, 10-17
Addition 4-10, 4-13
ADDITIVE 5-163, 5-164
ADEL() 6-3, 6-20
ADIR() 6-3, 6-21 - 6-22
    Returned File Attributes 6-21
Advantages, Clipper 3-2 - 3-3
AFIELDS () 6-3, 6-23 - 6-24
    FIELD() 6-58
AFILL() 6-3, 6-25
AINS() 6-3, 6-26
ALIAS
    FILE 4-5
    SKIP 5-170 - 5-171
ALIAS() 3-12, 6-3, 6-27
    UNLOCK 5-181, 10-3
ALLTRIM() 6-27.1
Alt-C (Terminate a Program) 5-167
    SETCANCEL() 6-117.1 - 6-117.2
Alt-D (Invoke the Clipper Debugger) 5-167, 6-27.2, 8-3
ALTD() 6-27.2, 8-3, F-3
Alternate File (.TXT) 4-4
ALTERROR.PRG F-1
ALTERROR.OBJ F-1
AmPm() J-12
.AND. 4-11
ANSI Terminal I/O Support 1-7, 2-2 9-3 - 9-4
ANSI.OBJ 9-3
    with SET COLOR 5-129
ANSI.SYS 9-3 - 9-4
APPEND C-1
    See also APPEND
        BLANK, ADD_REC()

APPEND BLANK 5-5, 5-32, 10-6, 10-13
NETERR() 6-100
APPEND FROM 5-6, 5-33 - 5-34, 10-6, 10-12 - 10-13
Appendices 1-3, A-I - J-15
ASCII Chart G-1
dBASE III PLUS Commands and Functions Not Supported C-1 - C-2
Description 1-3
Error Messages, Clipper D-1 - D-4
Error Messages, PLINK86-Plus E-6 - E-14
Error Messages, Run-time F-1
INKEY() Return Values G-2 - G-3
License Agreement B-1 - B-3
Nantucket Support A-1 - A-2
PLINK86-Plus Commands I-1 - I-3
Reserved Words H-1
Sample Programs J-1 - J-15
Warning Messages, PLINK86-Plus E-1 - E-5
Arrays 3-6, 4-9
ACHOICE() 6-15 - 6-18
ACOPY() 6-19
ADEL() 6-20
ADIR() 6-21 - 6-22
AFIELDS() 6-23 - 6-24
AFILL() 6-25
AINS() 6-26
ASCAN() 6-29
ASORT() 6-30
DECLARE 5-56 - 5-57.1
LEN() 6-84
PRIVATE 5-99
PUBLIC 5-101 - 5-102
ASC() 6-4, 6-28
ASCAN() 6-4, 6-29
ASCII 6-28, 6-78, 6-102, G-1
ASC() 6-28
Chart G-1
ASORT() 6-4, 6-30
Assembled Routines 1-5
Assembly Language 1-5, 11-3, 11-20 - 11-46
Autumn '86 Changes, Summary 11-44 - 11-46
Compatibility See EXTENDA.MAC
AX Values
__PARINFA 11-36
__PARINFO 11-37
Extend Macros 11-24
EXTENDA.INC 11-20, 11-25 - 11-26, 11-28 - 11-32
EXTENDA.MAC 11-20, 11-24, 11-46
Macros 11-25
Functions 11-32 - 11-43
Interfacing with 11-20
Macros, Basic 11-26 - 11-27
Parameters 11-20
Sample Routines 11-20
Assign Keys
SET FUNCTION 5-149
SET KEY 5-152
ASSIST C-1
AT() 6-4, 6-31
RAT() 6-108
Attributes, DOS File 6-55
AUTOEXEC.BAT 2-3, 7-2, 7-13
Autumn '86 and Assembly Language 11-20, 11-24, 11-44 - 11-46
Autumn '86, MEMOEDIT() 6-91
AVERAGE 5-6, 5-35
AX Values, Assembly Language 11-36 - 11-37

B

BATCH H-1, I-1

Batch Files 2-4, 7-14 - 7-15

BEGIN SEQUENCE...
BREAK...END 5-35.1 - 5-35.2, F-3 - F-5

BEGINAREA 7-21, 7-25, H-1, I-1
BIN2I() 6-31.1
BIN2L() 6-31.2
BIN2W() 6-31.3
BOF() 6-4, 6-32
RECNO() 6-110
BREAK 5-35.1 - 5-35.2
Break Menu, Debugger 8-11 - 8-12
Breakpoints, Debugger 8-6, 8-11 - 8-12
BROWSE C-1
See also DBEDIT() 6-42 - 6-45
Buffers
DOS 9-2 - 9-4
Index 9-6, 9-7

C

C Language 1-5, 11-2 - 11-19
Compiling 11-5
EXTEND.H 11-3
Manifest Constants in 11-6
Functions 11-7 - 11-19
Interfacing with 11-3
Linking 11-5
Macros 11-6
Receiving Parameters 11-3
Returning Values 11-3 - 11-4
Sample Function 11-4
CALL 5-6, 5-35.3 - 5-35.6
WORD() 6-133
Called Programs 1-5, 5-35.4
CANCEL 5-6, 5-36, 5-103
CASE 5-64 - 5-65
CDOW() 6-4, 6-33
CHANGE C-1
Chapter Descriptions 1-2 - 1-3
Character String 3-7, 3-8, 4-7
CHR() 6-4, 6-34 - 6-35
CL.BAT 7-15
Class Descriptions, Error System F-6 - F-10
CLD.BAT 7-15, 8-3
CLEAR 5-6, 5-37
@...CLEAR 5-23
CLEAR ALL 5-6, 5-38
STORE 5-174
CLEAR FIELDS C-1
CLEAR GETS 5-6, 5-39
CLEAR MEMORY 5-6, 5-40
STORE 5-174
CLEAR TYPEAHEAD 5-7, 5-41
KEYBOARD 5-86
CLIPCOPY.BAT 2-3
Clipper Compiler 1-4, 7-2 - 7-7
Advantages 3-2 - 3-3
Command Line 3-4, 4-15, 7-2 - 7-4
Commands 5-1 - 5-186
Conventions 5-2 - 5-3
Summary 5-5 - 5-19
Error Messages D-1 - D-4
dBASE, Differences 3-16, C-1 - C-2
Documentation 2-3
Enhancements
Commands 3-10 - 3-11
Functions 3-12 - 3-14
Features 3-4 - 3-9, 10-2 - 10-3
Functions 6-1 - 6-134
Summary 6-2 - 6-13
Installation 2-3
Reader Comment Card ii
Registration Card ii, A-1
Serial Number A-1
Clipper Debugger 3-9, 6-109, 8-1 - 8-13
Alias Lists 8-4
Break Menu 8-11 - 8-12
Breakpoints 8-6, 8-11 - 8-12
Cancelling Input 8-3
Control Menu 8-5 - 8-6
DEBUG.OBJ 8-2

Display Menu 8-7 - 8-8
DOS Link 7-7 - 7-8, 8-3
DOS Shell 8-6
Function Keys 8-8
Help Menu 8-10
How to Use 8-3
Information Displays 8-5
Invoking 8-4
Linking 7-15 - 7-16, 8-3
Menu Bar Selections 8-5 - 8-13
Menus 8-5 - 8-13
Navigation 8-4 - 8-5, 8-10
PLINK86-Plus 8-3
Procedure Lists 8-4
Speed Keys 8-10
Using 8-3
Variable Menu 8-9
"Waiting" Message 8-4, 8-7
Watch Menu 8-3, 8-12 - 8-13
CLIPPER.LIB 6-2, 7-21, F-1, I-1
Clipper Utilities 3-15, 12-1 - 12-19
DBU 12-2 - 12-5
INDEX 12-10
LINE 12-11
MAKE 12-12 - 12-19
RL 12-6 - 12-9
SWITCH 6-51.1
CLOSE 5-7
ALL 5-42 - 5-43
ALTERNATE 5-42 - 5-43
DATABASE 5-42 - 5-43
FORMAT 5-42 - 5-43
INDEX 5-42 - 5-43
CLOSE PROCEDURE
Ignored 5-162
.CLP File ("Clip" File) 5-147, 7-3, 7-5 - 7-7, 7-20
CMONTH() 6-4, 6-36
CODE H-1
Code Segment
CODE 11-21, 11-46
PROG 11-21
COL() 6-4, 6-37
Color Table 5-129, 6-117.4
Command Line 3-4, 7-2 - 7-4
Continuing 4-15
Command Line Method, PLINK86-Plus 7-9 - 7-10
Command Processor, DOS 9-2
Command Syntax Errors E-6 - E-7
COMMAND.COM 5-118, 8-6, 9-2, F-8
Commands, Clipper 5-1 - 5-186
Conventions 5-2 - 5-3
Summary 5-5 - 5-19
Commands, PLINK86-Plus I-1 - I-3
Comment (#)
MAKE 12-15
PLINK86-Plus I-1
Comment (%)
PLINK86, Older Versions 7-14
Comment (*)
Clipper 5-11, 5-94
Comment (&&)
Clipper 5-11, 5-94
COMMIT 5-7, 5-44
Compatible Indexes, dBASE 4-3, 4-25
Compatibility of Local Area Networks with Clipper 10-4 - 10-5
Compile and Link Using a Batch Program 2-3, 7-15
Compiler, Overview 1-5 - 1-6
See also Clipper Compiler
Compiling 2-4, 5-62, 7-1 - 7-28
Batch Files 2-4, 7-14 - 7-15
.CLP File 7-5 - 7-7
Options 7-4 - 7-5
Single Program File 7-2 - 7-3
Compiling CALLed Programs 5-35.4
Compiling C Code 11-5
COMSPEC 5-118, 9-2
CONFIG.SYS 9-3 - 9-4, 9-7, 9-8 - 9-9

Constants 4-7
Context Specific Help 4-20
CONTINUE 5-7, 5-45
Continue Command Line 4-15
Control Menu, Debugger 8-5 - 8-6
COPY 5-7, 5-46 - 5-48, 10-12
FILE 5-7, 5-49
STRUCTURE 5-7, 5-50, 10-12
STRUCTURE EXTENDED 5-7, 5-51
Copyright ii
COUNT 5-8, 5-52, 10-6
cprintf() 11-5
CREATE 5-8, 5-53, 10-12
CREATE FROM 5-8, 5-54 - 5-55, 10-12
CREATE LABEL C-1
CREATE QUERY C-1
CREATE REPORT C-1
CREATE SCREEN C-1
CREATE VIEW C-1
CTOD() 6-4, 6-38 - 6-39
Ctrl-U (Undo in GETs) 3-6
Ctrl-Z (End-of-File Mark) 5-47
CURDIR() 6-39.1
Currency() J-11
Cursor On and Off 3-6
See also SET CURSOR
Custom Designed Help 3-6, 4-20

D

DATA H-1
Data Dictionary 5-54
Data Segment
DATA 11-21, 11-46
Database Errors F-6 - F-7
Database File (.DBF) 4-3
Database Modifications 3-16
DATE 4-2, 4-6
CDOW() 6-4, 6-33
CMONTH() 6-4, 6-36
CTOD() 6-4, 6-38 - 6-39
DATE() 6-5, 6-40
DAY() 6-5, 6-41
DOW() 6-5, 6-47
DTOC() 6-5, 6-48
DTOS() 6-5, 6-49
Format 5-134
MONTH() 6-10, 6-99
SET CENTURY 5-127
SET DATE 5-134 - 5-135
STR() 6-121- 6-122
YEAR() 6-13, 6-134
DATE() 6-5, 6-40
DAY() 6-5, 6-41
STR() 6-121 - 6-122
Days() J-13
dBASE III PLUS
Commands and Functions not Supported C-1 - C-2
Compatible Indexes (.NDX) 4-25
Compatibility, Added J-3 - J-6
Modification to Applications 3-16
DBEDIT() 5-86, 6-5, 6-39, 6-42 - 6-45.4
Active Keys 6-43
Modes 6-43
Return Values 6-43
Status Messages 6-44
Db_error() F-6
.DBF File (Database File) 4-3
DBF() J-3
DBFILTER() 6-45.1
DBRELATION() 6-45.2
DBRSELECT() 6-45.3
.DBT File (Memo Field File) 4-3
DBU Utility Program 3-15, 12-2 - 12-5
DEBUG 7-28, H-1, I-1
DEBUG.OBJ 7-15, 8-2

Debugger
See Clipper Debugger
EC Rainbow
See ANSI Terminal Support
DECLARE 3-10, 5-8, 5-56 - 5-57.1
Definitions
Metasymbol 5-4
Symbol 5-3
See also Glossary I - IX
DELETE 5-8, 5-58, 10-6, 10-13
DELETE FILE 5-9, 5-70
DELETED() 6-5, 6-46
DELIMITED
APPEND FROM 5-33 - 5-34
COPY 5-46 - 5-48
DESCEND() 6-46.1
Descriptions, Appendices 1-3
Descriptions, Chapter 1-2 - 1-3
DEVICE=ANSI.SYS 9-3
Diagnostic Errors, PLINK86-Plus E-14
Dim2() J-15
DIR 5-8, 5-59
Disclaimer ii
DISKSPACE() 6-46.2 - 6-46.3
HEADER() 6-72.1
DISPLAY 5-8, 5-60 - 5-61
FILES C-1
MEMORY C-1
STATUS C-1
STRUCTURE C-1
USERS C-1
Display Menu, Debugger 8-7 - 8-8
Dividing by Zero 4-24
Division 4-10, 4-13
DO 5-8, 5-62 - 5-63
PARAMETERS 5-98
CASE 5-8, 5-64 - 5-65
WHILE 5-9, 5-66 - 5-68, 6-54
Documentation 2-3
DOS 1-4, 1-7, 2-2, 3-2, 7-16, 9-1 - 9-9, 10-4, 10-5
Access 5-118
ANSI.SYS 9-3 - 9-4
Automatic Memory Allocation 9-4 - 9-5
Command Line 3-8
Commmand Processor 9-2
COMMAND.COM 5-118, 9-2
COMSPEC 5-118, 9-2
CONFIG.SYS 9-3 - 9-4, 9-7, 9-8 - 9-9
Environment 6-71.1, 9-2 - 9-3
File Attributes 6-55
File Functions 3-5
File Open Modes 6-61
File Pointer, Moving 6-68
Files and Buffers 9-2 - 9-3
Linker 7-7 - 7-8
Linker, /SEGMENTS Options 7-8
Read-only Attribute 10-12
"SET" Command 9-4 - 9-5
DOSERROR() 6-46.4, F-3
DOT.PRG 10-7
DOUBLE 5-30
DOW() 6-5, 6-47
DTOC() 6-5, 6-48
DTOS() 3-12, 6-5, 6-49
Dup_Chk() J-6

E

EDIT C-1
Editing Keys 4-27, 5-105, 6-90
EJECT 5-9, 5-69
PCOL() 6-10, 6-103
PROW() 6-11, 6-107
ElapTime() J-13
ELSE 5-79
ELSEIF 5-79
EMPTY() 3-12, 6-5, 6-50

End of File 6-5, 6-51
Ctrl-Z 5-47
ENDAREA 7-21, 7-24, 7-25, 7-28, H-1, I-1
ENDCASE 5-64 - 5-65
ENDDO 5-66 - 5-68
ENDIF 5-79
ENDTEXT 5-177
Enhancements, Clipper 3-10 - 3-14
Environment, DOS 9-2 - 9-3
EOF() 6-5, 6-51
Equal To 4-11
ERASE 5-9, 5-70
Error Messages
Clipper D-1 - D-4
PLINK86-Plus E-6 - E-14
Run-time F-1 - F-15
Error System F-1 - F-15
ALTD() F-3
BEGIN SEQUENCE... BREAK...END F-3 - F-5
Database Errors F-6 - F-7
DOSERROR() F-3, F-8
Error Class Descriptions F-6 - F-10
Error Functions, Overview F-1 - F-6
Expression Errors F-7 - F-8
Miscellaneous Errors F-8- F-9
Open Errors F-9
Other Errors F-10 - F-11
Parameters, Automatic F-1 - F-3
Print Errors F-9 - F-10
PROCNAME() F-3
Recovery Strategy F-3 - F-5
RETURN Argument F-5 - F-6
Runtime Error Messages, Summary F-12 - F-15
Undefined Errors F-10
ERROR() C-1
ERRORLEVEL H-1
ERRORLEVEL() 6-51.1
Errors
Command Syntax E-6 - E-7
Input Object File E-9
Intel Format Object File E-12 - E-13
Output File E-10
Miscellaneous E-11
PLINK86-Plus Diagnostic E-14
Program Structure E-14
Work File E-8
ERRORSYS.PRG F-1
ERRORSYS.OBJ F-1
Escape Keys 4-27, 5-106, 6-91
Examples, Help 4-21
Exclusive
SET EXCLUSIVE 5-15, 5-144
USE 5-184, 10-5
.EXE File (Executable File) 1-4 - 1-5, 1-6, 7-2, 7-7, 7-18, 7-22
Executable File (.EXE) 1-4 - 1-5, 1-6, 7-2, 7-7, 7-18, 7-22
Executing Clipper-Compiled Programs 1-6, 1-7, 2-2
Excluding Memory 9-8
EXIT
DO WHILE 5-9, 5-66 - 5-68
FOR...NEXT 5-9, 5-74 - 5-75
EXP() 6-6, 6-52
SET DECIMALS 5-136
Expanded Memory 9-4 - 9-5, 9-7
Exponentiation 4-10, 4-13
EXPORT TO C-1
Expr_error() F-7, F-8
Expression Errors F-7 - F-8
Expressions 3-4, 4-14
Extend Functions Summary 11-3
Extend Macros 11-24
Extend System 11-2 - 11-46
Assembly Language Functions 11-32 - 11-43
C Interface Functions 11-7 - 11-19

Extended File, Structure 5-51
EXTEND.H 11-3, 11-6
EXTEND.LIB 6-2
EXTENDA.INC 11-20, 11-25 - 11-26, 11-28 - 11-32
EXTENDA.MAC 11-20, 11-24, 11-25, 11-46
Extended Character Set, IBM 6-34
EXTERNAL 3-10, 5-9, 5-71, 11-2

F

FCLOSE() 3-12, 6-6, 6-53
FCOUNT() 3-12, 6-6, 6-54
FCREATE() 3-12, 6-6, 6-55 - 6-56
DOS File Attributes 6-55
Features, Clipper 3-4 - 3-9, 10-2 - 10-3
FERROR() 3-12, 6-6, 6-57
FIELD() 6-6, 6-58
FIELDNAME() 6-6, 6-58
Fields 4-2, 4-6
APPEND FROM 5-32
Character 4-6
COPY 5-46 - 5-48
COPY STRUCTURE 5-50
COPY STRUCTURE EXTENDED 5-51
Date 4-2, 4-6
Fieldnames 4-6
JOIN 5-84 - 5-85
Logical 4-2, 4-6
Memo 4-2, 4-6
Numeric 4-2, 4-6
TOTAL 5-178 - 5-179
FIL_LOCK() 10-9, 10-13, 10-15
File 4-3 - 4-4, 9-2, 9-3, 9-8 - 9-9
Alternate (.TXT) 4-4
Attributes, DOS File 6-55
"Clip" (.CLP) 5-147, 7-3, 7-5 - 7-7, 7-20
Database (.DBF) 4-3
Executable (.EXE) 7-2, 7-7, 7-18, 7-22
Format (.FMT) 4-4, 5-147, 7-2
Functions, Assembly Language 11-32 - 11-43
Functions, C Language 11-7 - 11-19
Sample C Function 11-4
Functions, DOS File 3-5
Index (.NDX, .NTX) 4-3
I/O, Low-level J-1, J-10 - J-11
Label Form (.LBL) 4-4
"Link" (.LNK) 7-10 - 7-13, 7-18, 7-20, 7-22, 7-23, 7-24, 7-25
Memo Field (.DBT) 4-3
Memory (.MEM) 4-3
Open at One Time 9-2, 9-3, 9-8 - 9-9
Open Modes, DOS 6-61
Overlay, External (.OVL) 7-22
Procedure (.PRG) 4-4, 5-162, 7-2, 7-15, 7-20
Program (.PRG) 4-4, 5-162, 7-2, 7-15, 7-20
Report Form (.FRM) 4-4
Temporary (.$$$) 7-5
FILE I-1
FILE() 6-6, 6-59
File Lock 10-2, 10-6, 10-11
File Server 10-2
Filenames in Commands 4-3
FkLabel() J-3
FkMax() J-3
FIND 5-9, 5-72 - 5-73
FLOCK() 6-6, 6-60, 10-2, 10-4, 10-8, 10-13
FOPEN() 3-12, 6-6, 6-61 - 6-62
DOS File Open Modes 6-61
FOR
AVERAGE 5-35
COPY 5-46 - 5-48
COUNT 5-52

DELETE 5-58
DISPLAY 5-60 - 5-61
JOIN 5-84 - 5-85
LABEL FORM 5-87 - 5-88
LIST 5-89 - 5-90
LOCATE 5-91
RECALL 5-107
REPLACE 5-111 - 5-112
REPORT FORM 5-113 - 5-114
SORT 5-172 - 5-173
SUM 5-176
TOTAL 5-178 - 5-179
FOR...NEXT 3-10, 5-9, 5-74 - 5-75, 6-54
Format File (.FMT) 4-4, 5-147, 7-2
.FMT File (Format File) 4-4, 5-147, 7-2
.FRM File (Report Form File) 4-4
FOUND() 6-7, 6-63
See also CONTINUE, FIND, LOCATE
FREAD() 3-12, 6-7, 6-64 - 6-65
BIN2I() 6-31.1
BIN2L() 6-31.2
BIN2W() 6-31.3
FREADSTR() 3-12, 6-7, 6-66 - 6-67
BIN2I() 6-31.1
BIN2L() 6-31.2
BIN2W() 6-31.3
Free Memory Pool 9-4 - 9-5
FROM
APPEND 5-33 - 5-34
CREATE 5-54 - 5-55
RESTORE 5-115
RESTORE SCREEN 5-116
UPDATE 5-182 - 5-183
.FRM File (Report Form File) 4-4, 12-6
FSEEK() 3-12, 6-7, 6-68 - 6-69
Methods of Moving the DOS File Pointer 6-68
Full-Screen Editing Keys 5-105
Full-Screen Escape Keys 5-106
Full-Screen Mode Keys 5-106
Full-Screen Navigation Keys 4-26, 5-105
FUNCTION 5-9, 5-76 - 5-77
Function Key Mappings 5-149
Functions 3-4, 4-15, 6-1 - 6-134
Command Line 3-4
Error System, Overview F-1 - F-6
Formatting 3-8
Summary 6-3 - 6-13
See also User-Defined Function
FWRITE() 3-12, 6-7, 6-70 - 6-71
I2BIN() 6-72.1
L2BIN() 6-81.1

G

Generalized Help 4-22
GET Option (@ Command) 5-25 - 5-29
Terminating keys 5-131
READ 5-11, 5-104 - 5-106
GET_CHAR 11-25
GET_DATESTR 11-25
GET_DBL 11-25
GET_INT 11-25
GET_LOGICAL 11-25
GET_LONG 11-25
GET_PCOUNT 11-25
GET_PTYPE 11-25
GETE() 6-71.1
GO/GO TO 5-9, 5-78, 10-6
Greater Than 4-11
Greater Than or Equal To 4-11

H

HARDCR() 3-12 - 3-13, 6-7, 6-72
HEADER() 6-72.1

HEIGHT H-1, I-1
HELP C-1
Help
Context Specific 4-20
Custom Designed 3-6, 4-20
Examples 4-21
Generalized 4-22
Parameters 4-20
Help Menu, Debugger 8-10

I

I2BIN() 6-72.1
IBM Extended Character Set 6-34
IF...ELSE...ENDIF 5-9, 5-79 - 5-79.1
IF ERRORLEVEL 7-14 - 7-15
IF()/IIF() 6-7, 6-73 - 6-74
IMPORT FROM C-1
INDEX 5-10, 5-80 - 5-82, 10-7 - 10-8
Index Buffers 9-6, 9-7
Index File
.NDX (dBASE Compatible) 4-3, 4-25
.NTX (Clipper Default) 4-3
INDEX ON 10-12, 11-2
INDEX.PRG 2-4, 12-10
INDEX Utility Program 2-4, 12-10
INDEXEXT H-1
INDEXEXT() 3-13, 6-7, 6-75
INDEXKEY() 3-13, 6-7, 6-76
INDEXORD H-1
INDEXORD() 3-13, 6-8, 6-77
Information Displays, Debugger 8-5
INKEY() 3-5, 6-8, 6-78 - 6-79
LASTKEY() 6-82
NEXTKEY() 6-102
SET KEY 5-152 - 5-153
INKEY() Return Values
Cursor Keys G-3
Function Keys G-2
INPUT 5-10, 5-83
LASTKEY() 6-82
SET CONSOLE 5-132
Input Object File Errors E-9
INSERT C-1
Installing Clipper 2-1, 2-3
INT() 6-8, 6-80
Intel Format Object File Errors E-12
Interactive Link Method, PLINK86-Plus 7-9
Interface, Screen 3-5
Interface, User J-6 - J-10
Interfacing with Assembly Language 11-20
Interfacing with C 11-3
Interpreter, Overview 1-4 - 1-5
Invoking the Debugger 8-5 - 8-6
ISALPHA() 6-8, 6-80.1
ISCOLOR() 6-8, 6-81
ISLOWER() 6-81.1
ISPRINTER() 6-81.2
ISUPPER() 6-81.3

J

JOIN 5-10, 5-84 - 5-85
JOIN...TO 10-12

K

KEYBOARD 3-10, 5-10, 5-86, 6-34
Keyboard Control 3-5
Keys 4-26 - 4-27, 5-106
Assign 5-149, 5-152
Editing 4-27, 5-105
Escape 4-27, 5-106
Mode 4-27, 5-106
Navigation,

Full-Screen 4-26, 5-105
SET FUNCTION 5-149
SET KEY 5-152
Keywords 4-15

L

L2BIN() 6-81.1
LABEL FORM 5-10, 5-87 - 5-88, 10-12, 11-2, 12-6, 12-9
IIF() 6-73
SET MARGIN 5-154
See also RL Utility Program
Label Form File (.LBL) 4-4
LAN 10-1 - 10-17
Languages, Other 1-5
LASTKEY() 3-13, 6-8, 6-82
INKEY() 6-78
NEXTKEY() 6-102
LASTREC() 6-8, 6-83
HEADER() 6-72.1
.LBL File (Label Form File) 4-4
LEFT() 6-83.1
LEN() 6-8, 6-58, 6-84
LenNum() J-14
Less Than 4-11
Less Than or Equal To 4-11
LIBRARY I-1
Library, Clipper
See CLIPPER.LIB
License Agreement B-1 - B-3
Light-bar Menus 3-5
Line Number 7-4, 7-5, 8-2, 12-11
No Line Numbers 7-4
Suppress 7-4, 7-5
LINE Utility program 3-15, 12-1, 12-11
"Link" File (.LNK) 7-10 - 7-13, 7-18, 7-20, 7-22, 7-23, 7-24, 7-25
Linker 7-7 - 7-12
Linking 1-5, 7-7 - 7-28, 8-3
Batch Files 7-14 - 7-15
Clipper Debugger 7-15 - 7-16, 8-3
C Routines 11-5
DOS Linker 7-7 - 7-8
Errors 1-6
Optional Function Libraries 7-17
Optional Screen Drivers 7-16
PLINK86-Plus 7-9 - 7-12
See also Overlays
LIST 5-10, 5-89 - 5-90
LIST FILES C-1
LIST HISTORY C-1
LIST MEMORY C-1
LIST STATUS C-1
LIST STRUCTURE C-1
.LNK File ("Link" File) 7-10 - 7-13, 7-18, 7-20, 7-22, 7-23, 7-24, 7-25
LOAD C-1
Local Area Network 10-1 - 10-17
LOCATE 5-10, 5-45, 5-91
Lock Records and Files 10-5, 10-6
See also FLOCK(), FIL_LOCK(), RLOCK(), REC_LOCK()
LOCK() 3-13, 6-11, 6-112, 10-4
Locking 10-5
LOCKS.PRG 10-9, 10-13 - 10-17
LOG() 6-8, 6-85
EXP() 6-52
SET DECIMALS 5-136
Logical Operators 4-11
LOGOUT C-1
LOOP 5-67
Low-level File I/O J-1, J-10 - J-11
LOWER() 6-9, 6-86
UPPER() 6-131
LOWERCASE I-1

LTRIM() 6-9, 6-87
ALLTRIM() 6-27.1
TRIM() 6-128
LUPDATE() 6-87.1

M

Macro(&) 3-4, 4-17, 5-67, 5-72, 5-150
Special : MAKE Utility 12-15 - 12-19
($) 12-15, 12-16
($*) 12-17, 12-18
($**) 12-19
($@) 12-19
Macros, Assembly Language 11-26 - 11-27
Macros, C Language 11-6
MAKE Utility Program 3-15, 12-12 - 12-19
Comments 12-15
Description File 12-13
Inference Rules 12-17 - 12-19
Order of Rules 12-14
Macros 12-15 - 12-19
Sample System 12-15 - 12-16
Starting 12-14
MAKEDBU.BAT 12-2
MAP 7-25 - 7-26, I-2
Mathematical Operators
See Operators
MAX() 6-9, 6-88
MIN() 6-97
.MEM File (Memory File) 4-3
Memo Editor 3-8
Memo Field File (.DBT) 4-3
Memo Fields 3-8
REPLACE 5-111
Memo Handling 3-8
MEMOEDIT() 3-13, 6-9, 6-89 - 6-91.3
Editing Keys 6-90
Escape Keys 6-91
HARDCR() 6-72
Navigation Keys 6-90
READINSERT() 6-108.2
Status Messages 6-91.1
User Function Requests 6-91.2
MEMOLINE() 3-13, 6-9, 6-92 - 6-92.1
MLCOUNT() 6-98 - 6-98.1
MEMOREAD() 3-13, 6-9, 6-93
Memory
Automatic Allocation 9-4 - 9-5
Excluding 9-8
Expanded 9-4 - 9-5, 9-7
See also Overlay
Memory File (.MEM) 4-3
Memory Resident Programs 5-118
Memory Variables 3-6 - 3-7, 4-8 - 4-9, 5-4
@...SAY...GET 5-25
ACCEPT 5-31
Arrays 3-6
AVERAGE 5-35
Clipper 5-101 - 5-102
INPUT 5-83
List 4-8
MENU TO 5-92, 5-93
Names 4-8
PARAMETERS 5-96
PRIVATE 3-7, 4-8, 5-99, 5-117
PUBLIC 3-7, 4-9, 5-101
RELEASE 5-109
RESTORE 5-115
RESTORE SCREEN 5-116
RETURN 5-117
SAVE 5-120
SAVE SCREEN 5-121
Scope 4-8
STORE 5-174
SUM 5-176
WAIT 5-185
MEMORY() 3-13, 6-9, 6-94
MEMOTRAN() 3-13, 6-9, 6-95

MEMOEDIT() 6-89 - 6-91
MEMOWRIT() 3-13, 6-9, 6-96
Menu Bar Selections, Debugger 8-5 - 8-13
MENU TO 3-5, 3-10, 5-10, 5-24, 5-92
Active Keys 5-93
LASTKEY() 6-82
READVAR() 6-109
SET WRAP 5-169
MESSAGE C-1, 5-24
@...PROMPT 5-24, 5-92
MENU TO 5-24, 5-92
Metasymbol Definitions 5-4
MIN() 6-9, 6-97
MAX() 6-88
Miscellaneous Errors E-11, F-8 - F-9
MLCOUNT() 3-13, 6-10, 6-98 - 6-98.1
MLPOS() 6-98.1
Mode Keys 4-27, 5-106
Modes, DBEDIT() 6-43
MODIFY COMMAND C-1
MODIFY LABEL C-1
MODIFY QUERY C-1
MODIFY REPORT C-1
MODIFY SCREEN C-1
MODIFY STRUCTURE C-1
MODIFY VIEW C-1
Modulus 4-10
MONTH() 6-10, 6-99
STR() 6-121
Move Object File (Compiler Option) 7-4 - 7-5
Move Temporary File (Compiler Option) 7-5
Moving the DOS File Pointer 6-68
MS/PC DOS
See DOS
Multi_Form() J-7
Multiple Relations 3-7, 4-19
See also SET
RELATION
Multiplication 4-13

N

NANSIG (The Source) A-2
Nantucket Private Network (The Source) A-1, A-2
Nantucket News ii, A-1, A-2
Nantucket Software License Agreement B-1 - B-3
Nantucket Support A-1 - A-2
Navigation Keys, Full-Screen 4-26, 5-105
.NDX File (Index File, dBASE Compatible) 4-3, 4-25
NDX.OBJ 5-81, 6-75
.NTX File (Index File, Clipper Default) 4-3
NET_USE() 10-13, 10-14
NETERR() 3-7, 3-13, 5-32, 6-10, 6-100, 10-3, 10-4, 10-7, 10-8
NETNAME() 3-14, 6-10, 6-101
Network 3-7, 3-13, 10-1 - 10-17
Network Commands
APPEND BLANK 5-32
APPEND FROM 5-53
COPY 5-46 - 5-48
DELETE 5-58
RECALL 5-107
REINDEX 5-108
REPLACE 5-111 - 5-112
SET EXCLUSIVE 5-144
SET INDEX TO 10-8
SORT 5-172 - 5-173
UNLOCK 5-181
UPDATE 5-182 - 5-183
USE...EXCLUSIVE 5-184
ZAP 5-186
Network Functions

FLOCK() 6-60, 10-8
NETERR() 6-100
NETNAME() 6-101
RLOCK() 6-112
NEXT 5-9, 5-74, 5-75
NEXTKEY() 6-10, 6-102
NOBELL I-2
.NOT. 4-11
Not Equal To 4-11
NOTE 5-11, 5-94
Numeric Constants 4-7
Numeric Function J-14

O

.OBJ File (Object File) 1-5, 7-2, 7-7
Object File (.OBJ) 1-5, 7-2, 7-7
ON ERROR/ESCAPE/KEY C-1
On-line Support A-2
Open Errors F-9
Open Files
How to for Sharing 10-7, 10-8
Number of 9-8 - 9-9
Open_error() F-9
Operators 3-8, 3-9, 4-10 - 4-13
Addition 4-10
.AND. 4-11
Character 4-10
Division 4-10
Equal To 4-11
Exponentiation 4-10, 4-13
Greater Than 4-11
Greater Than or Equal To 4-11
Is Contained in the Set 4-11
Less Than 4-11
Less Than or Equal To 4-11
Logical 4-10, 4-11, 4-12, 4-13
Mathematical 4-10, 4-12, 4-13
Modulus 4-10
Multiplication 4-10, 4-13
.NOT. 4-11
Not Equal To 4-11
.OR. 4-11
Order of Evaluation 4-12
Relational 4-10, 4-12, 4-13
String 4-12, 4-13
Subset 4-11
Subtraction 4-10, 4-13
Optional Screen Drivers 7-16
ANSI.OBJ 7-16
Options, Compiler 7-4 - 7-5
.OR. 4-11
Order of Evaluation, Operators 4-12
Os() J-4
OTHERWISE 5-64
OUTPUT I-2
Output File Errors E-10
Overlay 1-6, 7-7, 7-18 -7-28, E-14
Area 7-18, 7-21
Creation 7-20
Debugging 7-28
Description 1-6, 7-18
Design 7-19
Errors, Structure E-14
External 7-22 - 7-23
Internal 7-22 - 7-23
.LNK File 7-20, 7-22, 7-23, 7-24
Management 7-25
Memory 7-18, 7-19, 7-21, 7-22, 7-23
Multiple Areas 7-22
Nested 7-23 - 7-28
Size 7-23, 7-27
Structure Errors E-14
Overlay File, External (.OVL) 7-22
OVERLAY PROG 7-14, 7-20, 7-21
OVERLAY.LIB 7-21, I-1
.OVL File (Overlay File, External) 7-22

P

PACK 5-11, 5-95, 10-6, 10-13
PARAMETERS 3-8, 5-11, 5-96 - 5-98
Parameters 3-8, 5-35.3, 5-57, 5-62, 11-3, 11-20
Parameters, Error System F-1 - F-3
Parameters, Help 4-20
Path Command Line 5-157
Path List Separator (;) 5-157
Pause (Compiler Option) 7-5
PC/MS-DOS
  See DOS
PCOL() 6-10, 6-103
PCOUNT() 3-14, 5-96, 6-10, 6-104
PHILES.PRG J-10 - J-11
PICTURE 5-5, 5-25 - 5-29
  Function Symbols 5-26
  Template Symbols 5-27
  TRANSFORM() 6-126, 6-127
PLINK86-Plus 7-8 - 7-28, 8-3, 9-2, E-1 - E-14, I-1 - I-3
  Batch Files 7-14 - 7-15
  Clipper Debugger 8-3
  Command and Syntax Errors E-6 - E-7
  Command Line Link Method 7-9
  Commands I-1 - I-3
  Diagnostic Errors E-14
  Error Messages E-6 - E-14
  Input Object File Errors E-9
  Intel Format Object File Errors E-12 - E-13
  Interactive Link Method 7-9 - 7-10
  .LNK File Link Method 7-10 - 7-12
  Miscellaneous Errors E-11
  Output File Errors E-10
  Program Structure Errors E-14
  Syntax Errors E-6 - E-7
  Tips 7-14
  Version 7-14
  Warning Messages 7-12, E-1 - E-5
  Work File Errors E-8
  See also Linking, Overlays
Predefined C Interface Macros 11-8
.PRG File (Program/Procedure File) 4-4, 5-162, 7-2, 7-15, 7-20
Print Errors F-9 - F-10
Print_error F-9
PRIVATE 5-11, 5-99, 5-101
  ALL 5-99
  EXCEPT 5-99
  LIKE 5-99
  PUBLIC 5-99
Private Memory Variables 5-40, 5-109
PROCEDURE 5-11, 5-100
Procedure File (.PRG) 4-4, 5-162, 7-2, 7-15, 7-20
PROCFILE H-1
PROCLINE() 3-14, 6-10, 6-105
PROCNAME() 3-14, 6-10, 6-106, F-3
PROG Code Segment 11-21
Program File (.PRG) 4-4, 5-162, 7-2, 7-15, 7-20
Program Overlays
  See Overlays
Program Structure Errors E-14
Programming for a Local Area Network 10-5
Programs, Sample J-1 - J-15
PROMPT
  MENU TO 5-92, 5-93
PROW() 6-11, 6-107
PUBLIC 5-11, 5-101, 5-102
PUBLIC Clipper 3-9, 3-16, 5-102
Public Memory Variables 5-40, 5-109

Q

QUIT 5-11, 5-103

R

Rainbow, DEC
  See ANSI Terminal Support
RANGE 5-25, 5-27
RAT() 6-11, 6-108
READ 5-11, 5-104 - 5-106
  CLEAR 5-104
  CLEAR ALL 5-104
  CLEAR GETS 5-104
  Full-Screen Editing Keys 5-105
  Full-Screen Escape Keys 5-106
  Full-Screen Mode Keys 5-106
  Full-Screen Navigation Keys 5-105
  GET 5-104
  LASTKEY() 6-82
  ROW() 6-114
  SAVE 5-120
  SET FORMAT 5-147, 5-148
  UPDATED() 6-130
  VALID 5-104
READ_ME.1ST ii, 2-3
Reader Comment Card ii
READEXIT 6-108.1, H-1
READINSERT 6-108.2, H-1
ReadKey() J-5
READVAR() 3-14, 6-11, 6-109
REC_LOCK() 10-9, 10-13, 10-16
RECALL 5-11, 5-107, 10-6, 10-13
RECCOUNT() 5-84, 6-8, 6-83
  HEADER() 6-72.1
RECNO() 6-11, 6-110
Record Lock 10-2, 10-11, 10-16
  See also RLOCK()
Recovery Strategy, Error System F-3 - F-5
RECSIZE() 6-110.1
  BIN2W() 6-31.3
  HEADER() 6-72.1
Registration Card ii
REINDEX 5-12, 5-108, 10-6, 10-13
  PACK 5-95
  SET UNIQUE 5-168
Relation
  DBRELATION() 6-45.2
  DBRSELECT() 6-45.3
  FOUND() 6-63
  Multiple 3-7
  See also SET RELATION
Relational Operators 4-10, 4-12, 4-13
Relative Seeking 3-7
RELEASE 5-12, 5-109
  Private Memory Variables 5-109
  PUBLIC Memory Variables 5-109
  STORE 5-174, 5-175
RENAME 5-12, 5-110
REPLACE 5-12, 5-111, 5-112, 10-6, 10-13, F-6
  UPDATE 5-182, 5-183
REPLICATE() 6-11, 6-111
  SPACE() 6-119
REPORT 11-2
REPORT FORM 5-12, 5-112, 5-113, 10-6, 10-12, 12-6, 12-8
  INKEY() 6-78, 6-79
  MEMOTRAN() 6-95
  SET MARGIN 5-154

TRANSFORM() 6-126, 6-127
Report Form File (.FRM) 4-4, 12-6
REPORT Utility Program 12-1
Reserved Word List H-1
RESTORE 5-12, 5-115
Private Memory Variables 5-115
PUBLIC Memory Variables 5-115
RESTORE FORM 10-12
RESTORE FROM 10-12
RESTORE SCREEN 3-5, 3-11, 5-12, 5-116, 10-10
RESTSCREEN() 6-111.1
SAVESCREEN() 6-114.1
RESUME C-1
RET_CHAR 11-25
RET_DATESTR 11-25
RET_DBL 11-25
RET_INT 11-25
RET_LOGICAL 11-25
RET_LONG 11-25
RETRY C-1
RETURN 5-12, 5-117
FUNCTION 5-76, 5-77
PRIVATE Memory Variables 5-99
PROCEDURE 5-100
STORE 5-174, 5-175
TO MASTER, not supported 5-117
RETURN Argument, Error System F-5 - F-6
RETURN TO MASTER 5-86, C-1
Return Values
DBEDIT() 6-43
INDKEY() G-2 - G-3
Returning Values from C 11-3 - 11-4
RIGHT() 6-111.2
RL Utility Program 3-15, 12-6 - 12-7, 12-8, 12-9
RLOCK() 6-11, 6-112, 10-2, 10-4, 10-8 - 10-9, 10-13
SET EXCLUSIVE 5-144
ROUND() 6-11, 6-113
Routines, Sample Assembly 11-20
ROW() 6-11, 6-114
RTRIM() 6-12, 6-128
ALLTRIM() 6-27.1
RUN 5-12, 5-118, 5-119, 9-6 - 9-8
Run-time 1-5, 1-6
Run-time Errors 1-6, F-1
See Error System
Runtime Environment 9-1, 9-6

S

Sample Assembly Routines 11-20
Sample C Functions 11-15
Sample Programs J-1 - J-15
SAVE 5-11, 5-120
SAVE SCREEN 3-5, 3-11, 5-13, 5-121, 10-10
RESTORE SCREEN 5-116
SAVE TO 5-13, 10-12
SAVESCREEN() 6-114.1
RESTSCREEN() 6-111.1
Scope 4-24
APPEND FROM 5-33, 5-34
AVERAGE 5-35
COPY 5-46, 5-47, 5-48
COUNT 5-52
DELETE 5-58
DISPLAY 5-60
LABEL FORM 5-87, 5-88
LIST 5-89, 5-90
RECALL 5-107
REPLACE 5-111
REPORT FORM 5-113
Screen Interface 3-5
SCROLL() 3-14, 6-11, 6-115
@...CLEAR 5-23

SDF
    APPEND FROM 5-33, 5-34
    COPY 5-46, 5-47, 5-48
SEARCH I-3
SECONDS() 3-14, 6-11, 6-116
    TIME() 6-125
Secs() J-13
SECTION 7-21, I-3
SECTION INTO 7-23, 7-24, 7-25, I-3
SEEK 5-13, 5-122, 10-6
    SET SOFTSEEK 5-166
Seeking, Relative 3-7
/SEGMENTS (DOS Link Option) 7-8
SELECT 5-13, 5-123
SELECT() 3-14, 6-12, 6-117
Serial Number, Clipper A-1
Service Mark Notices iii
SET C-2
SET ALTERNATE 5-13, 5-125
SET ALTERNATE TO 5-13, 10-12
SET BELL 5-13, 5-126
SET CARRY C-2
SET CATALOG C-2
SET CENTURY 5-13, 5-127, 6-48, 6-49
    DATE() 6-40
SET CLIPPER 9-6
    Buffers 9-7
    Excluding Memory 9-8
    Expanded Memory 9-7
    Memory Variables 9-6
    Open Files 9-8
    RUN Command 9-7
SET COLOR 5-13, 5-128, 5-129, 5-130
    SETCOLOR() 6-117.3 - 6-117.4
SET COLOR ON/OFF C-2
SET COMSPEC 5-118, 9-2
SET CONFIRM 5-14, 5-131
SET CONSOLE 5-14, 5-132
SET CURSOR 3-6, 3-11, 5-14, 5-133
SET DATE 5-14, 5-134
    DATE() 6-40
SET DEBUG C-2
SET DECIMALS 5-14, 5-136, 6-113
SET DEFAULT 5-14, 5-137
SET DELETED 5-14, 5-138
    LASTREC() 6-83
SET DELIMITERS 5-14, 5-139, 5-140
SET DEVICE 5-15, 5-25, 5-141
SET DO HISTORY C-2
SET ECHO C-2
SET ENCRYPTION C-2
SET ESCAPE 4-11, 5-15, 5-142
SET EXACT 5-15, 5-143
    ASCAN() 6-29
    STRTRAN() 6-123
SET EXCLUSIVE 5-15, 5-144, 10-3, 10-2, 10-5 - 10-7, 10-11
SET FIELDS C-2
SET FILTER 5-15, 5-145
    DBFILTER() 6-45.1
    LASTREC() 6-83
SET FIXED 5-15, 5-146, 6-113
SET FORMAT 5-15, 5-147
SET FUNCTION 5-15, 5-149
    WAIT 5-185
SET HEADING C-2
SET HELP C-2
SET HISTORY C-2
SET INDEX 5-16, 5-150, 10-8
SET INDEX TO 10-8, 10-12, 10-14
    INDEXKEY() 6-76
    INDEXORD() 6-77
SET INTENSITY 5-16, 5-151
SET KEY 3-11, 5-16, 5-152 - 5-153
    READVAR() 6-109
    SETCANCEL() 6-117.1 - 6-117.2
    UPDATED() 6-130
    WAIT 5-185
SET MARGIN 5-154
SET MEMO WIDTH C-2
SET MENUS C-2

SET MESSAGE 3-5, 5-16, 5-155
MENU TO 5-92, 5-93
SET OBJ=\CLIPPER 7-13
SET ORDER 5-16, 5-150, 5-156
SET PATH 5-16, 5-157, 5-158
SET PATH TO 4-15, 5-16
SET PRINT 5-16, 5-159
SET PRINTER 5-16, 5-160, 5-161
SET PRINTER TO 10-2 - 10-3
SET PROCEDURE 5-17, 5-162
SET PROCEDURE TO 7-3, 10-13
SET RELATION 3-7, 4-20, 5-17, 5-163, 5-164
DBRELATION() 6-45.2
DBRSELECT() 6-45.3
SET SAFETY C-2
SET SCOREBOARD 5-17, 5-163
SET SOFTSEEK 3-7, 3-11, 5-17, 5-166
SET STATUS C-2
SET STEP C-2
SET TALK C-2
SET TITLE C-2
SET TYPEAHEAD C-2, 5-17, 5-167
SET UNIQUE 5-17, 5-168
INDEX 5-80, 5-81, 5-82
SET VIEW C-2
SET WRAP 3-11, 5-17, 5-169
MENU TO 5-92, 5-93
SETCANCEL() 6-117.1 - 6-117.2
SETCOLOR() 6-117.3 - 6-117.4
SETPRC() 3-14, 5-25, 5-69, 6-12, 6-103, 6-118
PCOL() 6-103
PROW() 6-107
Shared mode 10-2, 10-7, 10-8, 10-9, 10-11
Skeleton, DIR 5-59
SKIP 5-17, 5-170, 5-171
BOF() 6-32
RECNO() 6-110
Software License Agreement B-1, - B-3
SORT 5-18
SORT...TO 10-12
SOUNDEX() 6-118.1
Source, The 2-2, A-1, A-2
SPACE() 6-12, 6-119
REPLICATE() 6-111
Specifications, Technical 4-2
SQRT() 6-12, 6-120
Status Messages, DBEDIT() 6-44
Status Messages, MEMOEDIT() 6-91.1
STORE 5-18, 5-174, 5-175
STR() 6-121
LTRIM() 6-87
String Functions J-11 - J-12
String Operators 4-12
Strings 4-7
STRTRAN() 6-12, 6-123
StrZero() J-11
Structure of an Extended File 5-51
STUFF() 6-123.1 - 6-123.2
Subset 4-11
SUBSTR() 6-12, 6-124
Subtraction 4-10, 4-13
SUM 5-18, 5-176, 10-6
SUMMER87 H-1
Support, Nantucket A-1 - A-2
SWITCH.EXE 6-51.1
Symbols, Syntax 4-15 - 4-16, 5-3
Syntax
Errors E-6 - E-7
Filenames 5-2
Keynames 5-2
Keywords 5-2
Metasymbol Definitions 5-4
Procedures 5-2
Symbol Definitions 4-15 - 4-16, 5-3
Typography 5-2
Syntax Rules 4-15 - 4-16
SYSTEM H-1
System Requirements 2-2

T

Technical Specifications 4-2
Technical Support A-1 - A-2
Temporary File (.$$$) 7-5
Terminate a Clipper-compiled Program 5-167
Texas Instruments Professional
See ANSI Terminal Support
TEXT 5-18, 5-177
Text Editor 1-4
Text File (.TXT) 5-180
The Source 2-2, A-1, A-2
Time Functions J-12 - J-14
TIME() 6-12, 6-125
SECONDS() 6-1116
Tips, PLINK86-Plus 7-14
TO
@...CLEAR 5-23
@...TO 5-30
ACCEPT 5-31
AVERAGE 5-35
COPY 5-46, 5-47, 5-48
COPY FILE 5-49
COPY STRUCTURE 5-50
COPY STRUCTURE EXTENDED 5-51
COUNT 5-52
FOR...NEXT 5-74, 5-75
INDEX 5-80, 5-81, 5-82
INPUT 5-83
JOIN 5-84, 5-85
RENAME 5-110
SAVE 5-120
SAVE SCREEN 5-121
SET ALTERNATE 5-125
SET COLOR 5-128, 5-129, 5-130
SET DELIMITERS 5-139, 5-140
SET DEVICE 5-141
SET FILTER 5-145
SET FUNCTION 5-149
SET INDEX 5-150
SET KEY 5-152, 5-153
SET MESSAGE 5-155
SET ORDER 5-156
SET PATH 5-157, 5-158
SET PRINTER 5-160, 5-161
SET PROCEDURE 5-162
SET RELATION 5-163, 5-164
SET TYPEAHEAD 5-167
SORT 5-172, 5-173
STORE 5-174, 5-175
SUM 5-176
TOTAL 5-178, 5-179
WAIT 5-185
TO FILE
DISPLAY 5-60, 5-61
LABEL FORM 5-87, 5-88
LIST 5-89, 5-90
REPORT FORM 5-113, 5-114
TEXT 5-177
TYPE 5-180
TO PRINT
DISPLAY 5-60, 5-61
LABEL FORM 5-87, 5-88
LIST 5-89, 5-90
REPORT FORM 5-113, 5-114
TEXT 5-177
TYPE 5-180
TONE() 6-125.1 - 6-125.2
TOTAL 3-7, 5-18, 5-178 - 5-179, 10-6
SUM 5-176
TOTAL...TO 10-12
Trademark Notices iii
TRANSFORM() 6-12, 6-126 - 6-127
Functions 6-126
Templates 6-127
TRIM() 6-12, 6-128
INDEX 5-80, 5-81 - 5-82
LTRIM() 6-87
Tstring() J-14

.TXT File (Alternate File) 4-4
.TXT File (Text File) 5-180
TYPE 5-18, 5-180, 10-12
TYPE() 6-12, 6-129 - 6-129.1
Return Values 6-129
Typography 5-2

U

UDF
See User-Defined Function
Undef_error() F-10
Undefined Errors F-10
Undo in GETs (Ctrl-U) 3-6
UNIQUE
INDEX 5-80, 5-81, 5-82
REINDEX 5-108
UNLOCK 5-18, 5-181, 10-3, 10-9, 10-11
LOCK() 6-112
UPDATE 5-18, 5-182 - 5-183
UPDATE ON 3-7, 10-6, 10-13
UPDATE...FROM 10-12
UPDATED() 3-14, 6-13, 6-130
LASTKEY() 6-82
UPPER() 6-13, 6-131
LOWER() 6-86
UPPERCASE I-3
USE 5-19, 5-184, 10-3, 10-4, 10-7, 10-8, 10-11
NETERR() 6-100
USED() 6-131.1
USE...EXCLUSIVE 5-184, 10-3, 10-11, 10-13
SET EXCLUSIVE 5-144
NETERR() 6-100
USE...INDEX 5-184
INDEXKEY() 6-76
INDEXORD() 6-75
USED() 6-131.1
User Interface J-6 - J-10
User Function Requests, MEMOEDIT() 6-91.2
User-Defined Function 3-4, 4-18, 5-96- 5-97, 7-19, 10-9
@...SAY...GET 5-28
EditMemo() Example 6-91
EditorMemo() Example 5-119
FUNCTION 5-76 - 5-77
RETURN 5-117
See also Extend System, FIL_LOCK(), REC_LOCK()
Utilities, Clipper 3-15, 12-1 - 12-19
DBU 12-2 - 12-5
INDEX 12-10
LINE 12-11
MAKE 12-12 - 12-19
RL 12-6 - 12-9

V

VAL() 6-13, 6-132
VALID 3-6, 3-10, 5-5, 5-25, 5-27 - 5-28
LASTKEY() 6-82
UPDATED() 6-130
Validation of GETs 3-6
ValidTime() J-8
Variable Menu, Debugger 8-9
VERBOSE I-3
Version, PLINK86-Plus 7-14
Version() J-5

W

WAIT 5-19, 5-185
LASTKEY() 6-82
SET CONSOLE 5-132
Wait State 3-5, 5-152, 6-82

"Waiting" Message,
Debugger 8-4, 8-7
ANG PC
See ANSI Terminal
Support
Warning Messages,
PLINK86-Plus E-1 - E-5
Warranty B-1, B-2
Watch Menu, Debugger 8-3, 8-12 - 8-13
WHILE
APPEND FROM 5-33, 5-34
AVERAGE 5-35
COPY 5-46, 5-47, 5-48
COUNT 5-52
DO 5-62 - 5-63
DELETE 5-58
DISPLAY 5-60 - 5-61
LABEL FORM 5-87 - 5-88
LIST 5-89 - 5-90
LOCATE 5-91
RECALL 5-107
REPLACE 5-111 - 5-112
REPORT FORM 5-113 - 5-114
SORT 5-172, 5-173
SUM 5-176
TOTAL 5-178
WIDTH I-3
WITH
JOIN 5-84, 5-85
WORD() 6-13, 6-133
Work Area Attributes 5-124
Work File Errors E-8
WORKFILE I-3

Y

YEAR() 6-13, 6-134
STR() 6-121

Z

ZAP 5-19, 5-186, 10-6, 10-13

Summer '87 Errata

Page 3-7 (More Memory Variables)

Replace the following paragraph:

Arrays...can contain 2048 elements.

with:

Arrays...can contain 4096 elements.

Page 5-6 (Summary of Commands)

Replace the syntax for APPEND FROM with:

APPEND [<scope>] [FIELDS <fields list>] FROM <file>/(<expC1>) [FOR <condition>] [WHILE <condition>] [SDF]/[DELIMITED [WITH BLANK/<delimiter>/(<expC2>)]]

Page 5-7 (Summary of Commands)

Replace the syntax for COPY FILE with:

COPY FILE <file1>.<ext>/(<expC1>) TO <file2>.<ext>/(<expC2>)

Page 5-8 (Summary of Commands)

Replace the syntax for DIR with:

DIR [<drive>:] [<path>\] [<skeleton>]//(<expC>)

Page 5-9 (Summary of Commands)

Replace the syntax for IF...ENDIF with:

IF...[ELSEIF]...[ELSE]...ENDIF

Page 5-9 (Summary of Commands)

Replace the syntax for ERASE/DELETE FILE with:

ERASE/DELETE FILE <filename>.<ext>/(<expC>)

Page 5-11 (Summary of Commands)

Replace the syntax for PUBLIC with:

PUBLIC <memvar/array list>/clipper

Page 5-11 (Summary of Commands)

Replace the syntax for PRIVATE with:

PRIVATE <memvar/array list>

Page 5-12 (Summary of Commands)

Replace the syntax for RENAME with:

RENAME <file1>.<ext>//(<expC2>) TO <file2>.<ext>//(<expC2>)

Page 5-14 (Summary of Commands)

Replace the syntax for SET DEFAULT with:

SET DEFAULT TO <drive>[:<path>]/(<expC>)

Page 5-16 (Summary of Commands)

Replace the syntax for SET PATH with:

SET PATH TO [<path list>]/(<expC>)

Page 5-17 (Summary of Commands)

Replace the syntax for SET RELATION with:

SET RELATION [ADDITIVE] TO [<key exp1>/RECNO()/<expN1> INTO <alias1>/(<expC1>)] [,TO <key exp2>/RECNO()/<expN2> INTO <alias2>/(<expC2>)]...

Page 5-18 (Summary of Commands)

Replace the syntax for TEXT...ENDTEXT with:

TEXT [TO PRINT] [TO FILE <file>/(<expC>)]...ENDTEXT

Page 5-18 (Summary of Commands)

Replace the syntax for TYPE with:

TYPE <file>.<ext>/(<expC>) [TO PRINT] [TO FILE <file>/(<expC>)]

Page 5-18 (Summary of Commands)

Replace the syntax for UPDATE with:

UPDATE ON <key exp> FROM <alias>/(<expC>) REPLACE <field1> WITH <exp1> [,<field2> WITH <exp2>]...[RANDOM]

Page 5-22 (@...BOX)

In the example section, remove the commas from the second line of the assignment string for the memory variable "single_double."

Page 5-25 (@...SAY...GET)

Change the syntax notation for the RANGE and VALID clauses of @...SAY...GET from:

[RANGE <expN1>, <expN2>] [VALID <expL>]

with:

[RANGE <expN3>, <expN4>]/[VALID <expL>]

where <expN3> is the lower range limit and <expN4> is the upper range limit.

Note that if both clauses are specified and the VALID clause precedes the RANGE clause, you will get a compiler error. If the VALID clause follows the RANGE clause, it generates a compiler error, but is ignored at runtime.

Page 5-28 (@...SAY...GET)

In the code example, replace the following line of code:

VALID n 0

with:

VALID number 0

Page 5-34 (APPEND FROM)

In the Usage section, replace the sentence:

If they are not you will get the error message "Type conflict in REPLACE (Q/A/I)?" when you APPEND FROM.

with:

If they are not, a Miscellaneous class runtime error will be generated with a "Type mismatch" info message when the APPEND FROM statement executes.

Page 5-42 (CLOSE)

Change the following Usage section paragraph from:

Clipper also closes files from a runtime error prompt. When you terminate program execution in response to a runtime error the following options close files.

- Quit option from (Q/A/I) prompt
- Pressing "N" from the Continue option closes all files and returns to the operation system.

to the following:

Clipper also closes files from the default runtime error functions and non-recoverable error prompts where you answer "N" to the Continue option. Refer to Appendix F, *The Runtime Error System*, for more information.

Page 5-44 (COMMIT)

Replace the Usage section with the following:

Under DOS 3.3 or greater, COMMIT flushes all Clipper buffers to DOS and then performs a solid-disk write. Under DOS 3.2 or less, COMMIT only flushes Clipper buffers to DOS.

Page 5-59 (DIR)

Replace the syntax for DIR with:

DIR [<drive>:] [<path>\] [<skeleton>]/(<expC>)

Pages 5-64, 5-68, 5-79 (ClipperCommands)

Add to the Usage section:

Note that END is now a reserved word and should not be used when naming memory variables.

Page 5-71 (EXTERNAL)

The Usage section should read:

Procedures, user-defined functions, and SET KEY procedures must be declared EXTERNAL if they are called with a macro and placed in overlays.

Page 5-85 (JOIN)

In the Example section, replace the JOIN statement with the following:

```
JOIN WITH Invoices TO Purchases;
FOR Last = Invoices->Last;
```

FIELDS First, Last, Invoices->Number,;
Invoices->Amount

Page 5-105 (READ)

In Table 5-7, replace the line:

At end of GET, cursor moves to next GET.

with:

Does not move cursor to next GET.

Page 5-122 (SEEK)

Add SET FILTER to the list of see also commands and functions.

Page 5-126 (SET BELL)

In the example code for SET BELL, add an APPEND BLANK statement immediately preceeding the READ statement.

Page 5-130 (SET COLOR)

In the example code, replace the following line of code:

IF getvar word

with:

IF getvar <> word

Page 5-139 (SET DELIMITERS)

Replace the syntax for SET DELIMITERS TO with:

SET DELIMITERS TO [<expC>/DEFAULT]

Page 5-174 (STORE)

In the usage section, replace the following sentence:

The (-v) switch changes this so that memory variables have precedence over field names.

with:

The (-v) switch changes this so that all references to variable identifiers specified without an alias refer to memory variables. All references to fields, therefore, must be proceeded by an alias identifier.

Page 6-3 (Summary of Functions)

Delete the repeated summary entry for ADEL() following ADIR().

Page 6-12 (Summary of Functions)

Add the following function between the entries for SQRT() and STRTRAN():

STR(<expN1>, [<expN2> [,<expN3>]])

Converts a numeric value to a character string.

Page 6-16 (ACHOICE())

Change the following Table 6-1 entries:

Ctrl-PgDn	First element
Ctrl-PgUp	Last element

to the following:

Ctrl-PgUp	First element
Ctrl-PgDn	Last element

Page 6-17 (ACHOICE())

Change the following Table 6-2 entries:

Ctrl-PgDn	First element
Ctrl-PgUp	Last element

to the following:

Ctrl-PgUp	First element
Ctrl-PgDn	Last element

Page 6-23 (AFIELDS())

In the example at the bottom of the page, change the variable name "ftype" to "flen" in order to correctly represent the argument as the field length.

Page 6-24 (AFIELDS())

In the example code, change the program line from:

@ 12, 0 SAY If (fld 0, fname[fld], "None selected")

to the following:

@ 12, 0 SAY If (fld <> 0, fname[fld], "None selected")

Page 6-25 (AFILL())

In the usage section, replace the entire paragraph with the following:

Note that AFILL() only evaluates the <exp> argument once and therefore it is not possible to increment values for the range of array elements specified.

Page 6-63 (FOUND())

In the example section, replace the following two separate lines of code:

```
* SEEK "100"
. . .
* LOCATE FOR Branch = "100"
```

with:

```
SEEK "100"
. . .
LOCATE FOR Branch = "100"
```

Page 6-64 (FREAD())

Add the following sentence after the first sentence of the Usage section:

After reading, the file pointer is located at the next unread character position.

In the arguments section, replace the phrase:

DOS pointer location

with:

DOS file pointer location

Page 6-66 (FREADSTR())

In the argument description for <expN2>, delete the sentence:

This can be a positive or negative number depending on the direction (forward or backward) you want read from the current pointer position.

Page 6-123 (STRTRAN())

Replace the last two sentences in the Usage section with:

SET EXACT has no affect on STRTRAN().

Change the STRTRAN() library reference from CLIPPER.LIB to EXTEND.LIB.

Page 7-4 (Compiler Options)

Add the compiler switch:

-v: All identifiers are assumed to refer to memory variables, as opposed to database fields, unless alisased.

Page 7-10 (Linking With PLINK86)

Replace the description for the command line argument "Libname" with the following paragraph:

"Libname" is the name of the required runtime library. By default PLINK86 will look for CLIPPER.LIB and OVERLAY.LIB. If, however, you specify any other libraries (i.e., EXTEND.LIB), you must specify CLIPPER.LIB explicitly. Failure to do so will result in warning errors 10 and 11 and/or runtime error 6000, stack underflow.

Replace the following command::

C> PLINK86 FI TEST, PROG1, PROG2, LIB CLIPPER EXTEND

with:

C> PLINK86 FI TEST, PROG1, PROG2 LIB CLIPPER EXTEND

Page 7-13 (Running PLINK86)

Add the command SET LIB=\CLIPPER below SET OBJ=\CLIPPER.

In reference to the display message after PLINK completes its linking, replace the sentence:

The number in parentheses is the executable file size.

to the following sentence:

The number in parentheses is the amount of memory required by the operating system before the linked program can be loaded for execution.

Pages 7-14, 7-20, 7-21, 7-24, 7-25 (Compiling and Linking)

References to PROG, $CONSTANTS should be changed to CODE, $CONSTANTS.

Page 9-6 (Modifying the Runtime Environment)

Add the SET CLIPPER parameter "Sn." The Sn parameter eliminates "snow" caused by some graphics adapters. Zero is the default and has the fastest screen writes. One will solve the "snow" problem if it occurs on your system.

Page 10-17

In the listing of the user-defined function Add_rec(), replace the program line:

ENDIF forever = (wait = 0)

with the following program lines:

ENDIF
forever = (wait = 0)

Page 12-11 (The Line Program)

LINE.EXE has an undocumented command line argument, /p. To correct the current syntax and argument descriptions, replace the command line syntax with the following:

C>LINE <filename> [/p]/[line number]

Page G-3 (Appendix G)

Change the numeric value for Ctrl-PgUp from 1 to 31.

Add the numeric value of 6 for End.

Page I-3 (PLINK86-Plus Commands)

The syntax for the WORKFILE command is: WORKFILE = [drive: <path>] <filename>. WORKFILE is used for temporary space when your machine does not have sufficient memory to produce an executable file (.EXE).

2

BUSINESS REPLY SERVICE
LICENCE NO. SG 595

NANTUCKET CORP. (EUROPE)
2 BLUECOATS AVE.
FORE ST.
HERTFORD SG 14 1PB
HERTS, ENGLAND, U.K.

Nantucket Support℠

Nantucket Support offers you a comprehensive service designed to provide solutions to your programming problems as well as technical information to expand your understanding of Clipper's innovative features. A yearly subscription fee of £99.00 entitles you to:

- unlimited inquiries to Nantucket Support
- a subscription to Nantucket News,™ our technical quarterly

You'll find Nantucket Support to be an indispensable service as you explore the Clipper environment with its unique programming advantages. Highly trained technicians provide rapid responses to your programming questions. And Nantucket News will provide valuable tips that refine your programming skills and enhance your programs. See Appendix A for more detailed information on Nantucket Support.

Nantucket News Only

You may also order the newsletter by itself without subscribing to support. The subscription fee is £20.00 per annum.

To subscribe to Nantucket Support or Nantucket News, complete and mail the attached card.

Clipper™

Serial # _____

NANTUCKET SUPPORT℠ SUBSCRIPTION

To subscribe to Nantucket's Clipper Support Plan, fill out this card and mail it in to receive the following:

Please check one: _____ TELEPHONE SUPPORT: £99.00—Unlimited calls may be placed to Nantucket Support without additional Charge. Includes subscription to Nantucket News.

_____ QUARTERLY NEWSLETTER: £20.00—Tips, documentation supplements, anomaly reports, and useful programs and functions will be sent to all subscribers.

Name (same as on registration) _____

Company _____

Address _____

_____

City _____ Country _____

Telephone _____ Purchase Date _____

I am enclosing:

☐ Cheque(U.K. only) ☐ International Money Order ☐ Access ☐ Visa ☐ American Express

Card # _____ Expiration Date _____

Authorized Signature _____

2

BUSINESS REPLY SERVICE
LICENCE NO. SG 595

NANTUCKET CORP. (EUROPE)
2 BLUECOATS AVE.
FORE ST.
HERTFORD SG 14 1PB
HERTS, ENGLAND, U.K.

2

BUSINESS REPLY SERVICE
LICENCE NO. SG 595

NANTUCKET CORP. (EUROPE)
2 BLUECOATS AVE.
FORE ST.
HERTFORD SG 14 1PB
HERTS, ENGLAND, U.K.

READER COMMENTS AND SUGGESTIONS

Nantucket Corporation believes your support is essential for complete customer satisfaction. We work constantly to enhance and improve our product. As a result, we periodically release new versions of both our software and the associated documentation.

Therefore, in order that both the software and the accompanying manual provide the most current information available, we would very much appreciate any comments you may have on the contents of the manual and product. Please suggest enhancements that will serve to improve the product for your use.

Space has been provided for you to record your comments and suggestions on two cards. Please complete and mail the bottom card that asks for your first impressions. Retain the middle card and return it with your comments once you become familiar with the product.

COMMENTS AND SUGGESTED ENHANCEMENTS

Please complete and return this card with your first impressions. Comment on both the product and the manual. Please offer any suggestions for future enhancements.

Reader's name _____

Company _____

Address _____

_____

City _____ Country _____

Telephone _____

READER'S COMMENTS ON CLIPPER™

_____

_____

_____

_____

_____

_____

_____

COMMENTS AND SUGGESTED ENHANCEMENTS

Please retain this card for future use. Return it with your comments and suggestions once you become familiar with the product and manual.

Reader's name _____

Company _____

Address _____

_____

City _____ Country _____

Telephone _____

READER'S COMMENTS ON CLIPPER™

_____

_____

_____

_____

_____

_____

_____